游戏设计与开发



DirectX 12 3D 游戏开发实战



[美] 弗兰克 · D. 卢娜（Frank D. Luna）著

王陈 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS


MERCURY
游戏设计与开发
DirectX 12 3D
游戏开发实战
3D GAME PROGRAMMING
[美] 弗兰克·D. 卢娜（Frank D. Luna）著
中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

版权信息

书名：DirectX 12 3D 游戏开发实战

ISBN：978-7-115-47921-1

版　　权

著　　　　[美] 弗兰克•D.卢娜（Frank D. Luna）

译　　　　王　陈

责任编辑　罗子超

人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

邮编　100164　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

读者服务热线：(010)81055410

反盗版热线：(010)81055315

版权声明

内容提要

Direct3D是微软公司DirectX SDK集成开发包中的重要组成部分，是编写高性能3D图形应用程序的渲染库，适用于多媒体、娱乐、即时3D动画等广泛和实用的3D图形计算领域。

本书围绕交互式计算机图形学这一主题展开，着重介绍Direct3D的基础知识和着色器编程的方法，并介绍了如何利用Direct3D来实现各种有趣的技术与特效，旨在为读者学习更高级的图形技术奠定坚实的基础。本书包括3部分内容。第一部分介绍必备的数学知识，涵盖向量代数、矩阵代数和变换等内容。这是贯穿全书的数学工具，是读者需要掌握的基础内容。第二部分重点介绍Direct3D的基础知识，展示用Direct3D来实现绘图任务的基本概念与技术，如渲染流水线、纹理贴图、混合、曲面细分等。第三部分则利用Direct3D来实现各种有趣的特效，如实例化与视锥体剔除、阴影贴图、环境光遮蔽等。

本书适合希望通过Direct3D来学习3D编程的C++中级程序员阅读，也可供已对Direct3D有一定了解或具有非DirectX API使用经验的3D程序员参考。

谨以此书献给我的侄辈们——

Marrick、Hans、Max、Anna、Augustus、Presley以及Elyse

前　　言

Direct3D 12是一款为运行在现代图形硬件上的各种Windows 10平台（Windows桌面版、手机版和Xbox One）编写高性能3D图形应用程序的渲染库。Direct3D也是一种底层库，这也就意味着此种应用程序接口（API）与其下层控制的图形硬件模块关系更为紧密[1]。Direct3D的主要用户大多来自游戏产业，他们驾驭Direct3D来构建更加高端的渲染引擎。同时，它亦应用于如医药产业、科学可视化以及虚拟建筑漫游等行业，用来实现高性能的3D图形交互功能。另外，由于当今每一部新的个人电脑都已配备了现代图形设备，因此，非3D应用也开始逐步把计算密集型的工作移交至显卡来执行，以充分发挥其中GPU（Graphics Processing Unit，图形处理器）的计算能力。这就是众所周知的GPU通用计算（general purpose GPU computing）技术。对此，Direct3D也提供了用于编写GPU通用计算程序的计算着色器API。尽管Direct3D 12程序通常以原生的C++语言进行编写，但SharpDX团队正在致力于.NET包装器版的开发，因此，用户也可以从托管应用程序中来访问这一强大的3D图形API。

本书围绕交互式计算机图形学这个主题展开，关注于通过Direct3D 12来进行游戏的开发。读者将从中学到Direct3D的基础知识以及着色器编程的方法。阅读完本书之后，读者就可以继续学习更加高级的图形技术了。本书共分为3个主要部分。第一部分讲解了本书后续要用到的数

学知识。第二部分展示如何用Direct3D来实现基本绘图任务，例如初始化Direct3D，定义3D几何图形，设置摄像机，光照，纹理，混合技术，模板技术，曲面细分技术，创建顶点、像素、几何图形以及计算着色器。第三部分则主要是利用Direct3D来实现各种有趣的技术与特效，例如动画角色网格、拾取技术、环境贴图、法线贴图、阴影贴图以及环境光遮蔽技术。

初学者最好按先后顺序通读全书。书中章节是按照由浅入深、逐步递进的顺序组织而成的。这样一来，读者便不会因过陡的学习曲线而如堕烟海。一般来讲，特定篇章中所用的技术与概念往往在之前的章节中有所交代。因此，读者最好在掌握了欲学习章节之前的所有内容后再继续前行。当然，有一定经验的读者可直接挑选感兴趣的部分进行阅读。

最后，部分读者可能会不禁琢磨：读完本书之后，究竟能够开发出何种类型的游戏来呢？这里对此给出的解释是：您最好亲自粗略地阅览此书，看看其中大概都在讲些什么内容。据此，基于本书所讲的技术知识再结合自己的聪明才智，至于能够开发出哪类游戏作品，想必这答案读者也就自会了然于胸了。

本书受众

本书主要适合以下3类读者：

1．希望通过Direct3D最新版本来学习3D图形学编程的C++中级程序员。

2．具有非DirectX API（如OpenGL）使用经验，并希望学习Direct3D编程方面知识的3D程序员。

3．具有一定的Direct3D使用经验，并希望学习Direct3D最新版本的程序员。

预备知识

需要强调的是，本书为重点介绍Direct3D 12、着色器编程以及3D游戏编程的读物，而并非是讨论一般计算机程序设计的读物。因此，读者需要具备下列预备知识：

1．高中程度的数学知识，比如代数、三角学以及（数学）函数等。

2．Visual Studio相关的使用技能，比如如何创建项目、为项目添加文件以及指定需要链接的外部库等。

3．中级C++编程技能以及数据结构知识，比如熟练地运用指针、数组、运算符重载、链表、继承、多态等。

4．熟悉使用Win32 API进行Windows编程还是很有必要的，可谓是学习本书的基础。但这一条并非是强制性要求，因为本书附录A中提供了Win32编程的相关入门知识。

需要配备的开发工具以及硬件环境

下面是进行Direct3D 12编程的必备条件：

1．Windows 10操作系统。

2．Visual Studio 2015开发环境或其后续版本。

3．一款支持Direct3D 12的显卡（本书中的演示程序都已通过Geforce GTX 760平台的测试）。

使用DirectX SDK文档以及SDK示例

Direct3D是一种规模庞大的API，将其所有的细节都在一本书中体现是不切实际的。因此，为了获得更为深入的API信息，学习DirectX SDK[2]文档的查阅方法势在必行。DirectX SDK在MSDN上的最新文档为《Direct3D 12 Programming Guide》，即《Direct3D 12编程指南》。

图1所示的是在线文档的截图。

DirectX文档涵盖了DirectX API的方方面面，因此，它是一种不可或缺的参考资料。然而，由于此文档对预备知识的讲解并不深入且假设读者对此有一定认识，因而导致它无法成为初学者最佳的学习工具。但是，随着DirectX每个新版本的发布，该文档也在日益完善中。

换言之，这个文档主要还是用作参考。假设用户碰到一个与DirectX有关的数据类型或函数，如函数`ID3D12Device::CreateCommittedResource`，并希望获取更多与之相关的信息，就可以方便地在该文档中搜索它，比如本示例中的函数（见图2），以得到更为细致的描述。

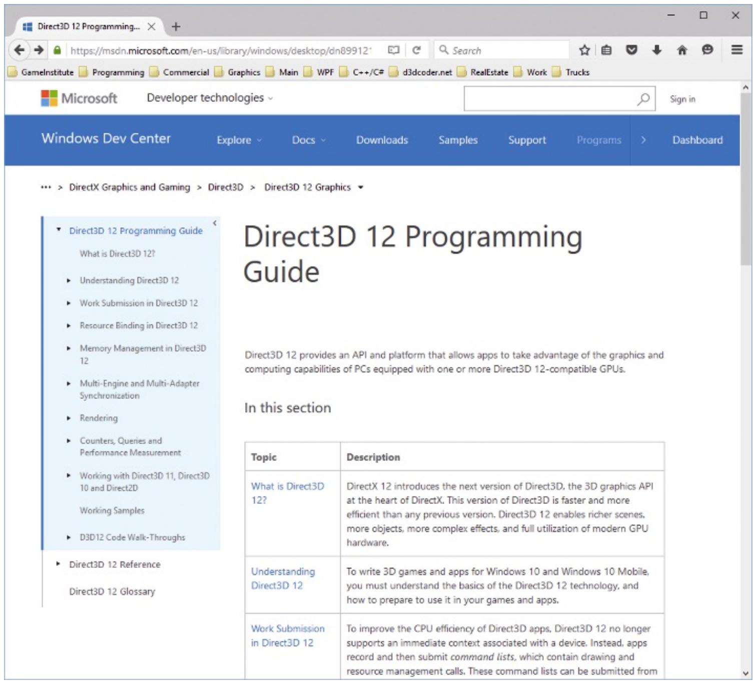


图1　DirectX文档中的《Direct3D 12编程指南》

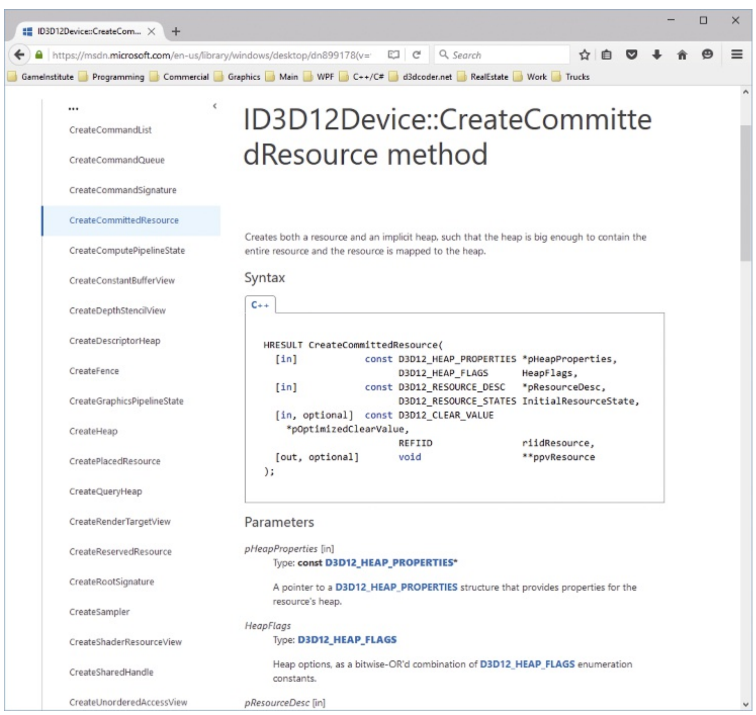


图2　获取函数的相关文档

注意 Note

在本书中，我们会不时地指导读者去阅览文档以获取更多的有关细节。

我们还建议读者研究一下官方提供的Direct3D 12演示程序。

微软官方可能还会在此陆续增添更多的例程。除此之外，读者还可以去NVIDIA、AMD以及Intel的官方网站上查找与Direct3D 12有关的示例。

明确学习目的

尽管我们努力遵循Direct3D 12的最佳实践，力图写出高效的代码，但本书中每个样例的主要目标还是为了阐述Direct3D中的基本概念以及演示图形编程技术。应当明确的是，写出最优代码并非本书最终目的，而且过分优化还可能导致原本意图明晰的代码变得含混不清，反而适得其反。希望读者将这一点铭记于心，尤其是在将书中例程代码合并到自己的项目中时，因此在此过程中，您可能为了追求程序更高的效率而重构代码。再者，为了把注意力集中在Direct3D API上，我们还在Direct3D之上构建了一层轻量级的框架。这就意味着我们很可能会在源代码中，以硬编码的数值与定义其他内容的方式来令程序得以运行。类似地，在大型的3D应用程序中，可能要在Direct3D的基础之上实现一款渲染引擎。但本书的主旨却是Direct3D API，而非设计渲染引擎。

例程与在线补充材料

读者可以登录本书的网站（www.d3dcoder.net和www.merclearning.com），以获取本书相关材料。在前者中，读者可以找到本书内所有例程的完整源代码以及项目文件。也可通过异步社区本书页面获取（www.epubit.com）。在大多数情况下，DirectX程序往往比较庞大，以至于不宜全部列入书中。因此，只得在书中嵌入与所讲内容密切相关的代码片段。为此，我们极力建议读者在学习相关的例程代码时去一睹它的全貌（为了便于读者学习，我们已将演示程序的规模尽量减小）。一般说来，在阅读过特定章节，并研究完所附演示代码后，读者应当能够自行独立地实现该章节中所述的例程。但事实上，一种更快捷的学习方法是在参考书籍和示例代码的同时，尝试着以自己的方式实现相关程序。

通过Visual Studio 2015安装演示项目[3]

通过双击项目文件（.vcxproj）或解决方案文件（.sln）就可以方便地打开本书的演示程序。接下来，我们将详述如何通过Visual Studio 2015 (VS15)以本书的例程框架从头开始创建并构建一个项目。在此，我们以第6章中的“Box”（立方体）演示程序为例。

下载本书的源代码

首先，读者需要下载本书所用的源代码并将其保存在硬盘的某个文件夹之中。为了便于讨论，假设这个文件夹的路径为C:\d3d12book。在这里可以看到一系列文件夹，其中含有对应章节的例程项目。读者可能会注意到有个名为“Common”的文件夹，其中包含所有演示项目中都要复用的公共代码。现在便可以在源代码文件夹中新建一个文件夹，用来存放我们自己的例程，例如C:\d3d12book\MyDemos。随后，我们将基于本书中的例程框架在该文件夹中创建一个新的项目。

注意 Note

事实上，读者自己设置的目录结构大可不必如此，这只不过是本书例程的结构而已。如果读者希望按自己的意愿来设置源代码文件，可以将演示项目放在任何地方，只要使Visual Studio能找到Common目录中的

源代码即可。

创建一个Win32项目

首先运行VS15，接着在主菜单中依次选择**File**（文件）→**New**（新建）→**Project**（项目），如图3所示。

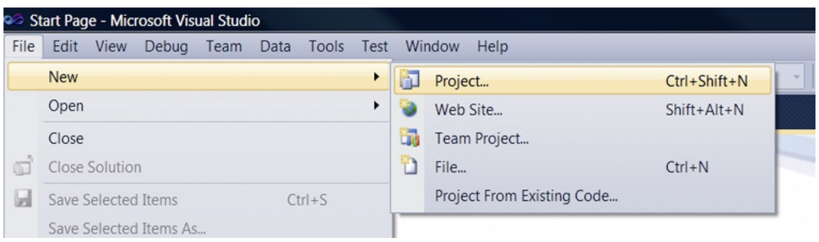


图3　创建一个新项目

在弹出的**New Project**（新项目）对话框（如图4所示）左侧Visual C++项目类型的树形控件中选择**Visual C++**→**Win32**，再于右侧选择**Win32 Project**（Win32项目）。接下来，给项目起个名称，并指定项目文件夹的保存位置。别忘了取消默认选中的**Create directory for solution**（为解决方案创建目录）复选框。随后单击**OK**（确定）按钮。

接着，又会弹出一个新的对话框。其左侧有**Overview**（概述）和**Application Settings**（应用程序设置）两个选项。选择**Application Settings**，便会出现如图5所示的对话框。在这里，需要确保选择**Windows application**（Windows应用程序）选项和**Empty project**（空项目）复选框，之后再单击**Finish**（完成）按钮。至此，我们已成功创建

了一个空的Win32项目，但在构建DirectX项目例程之前，我们还有一些事情需要做。

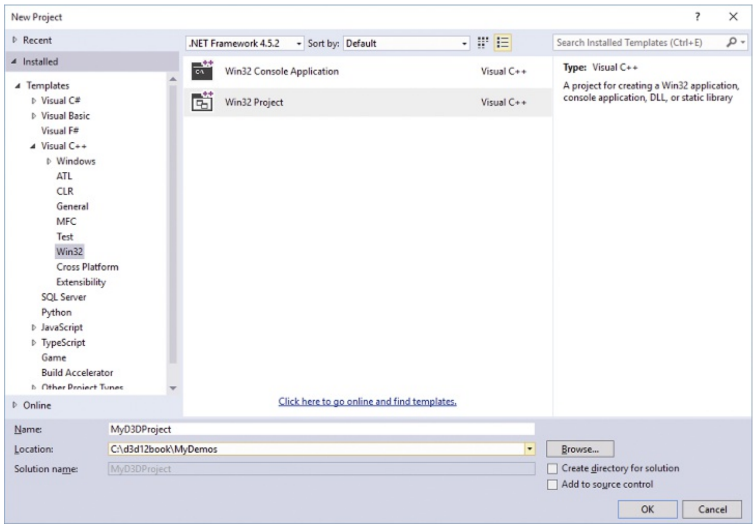


图4　新项目的相关设置

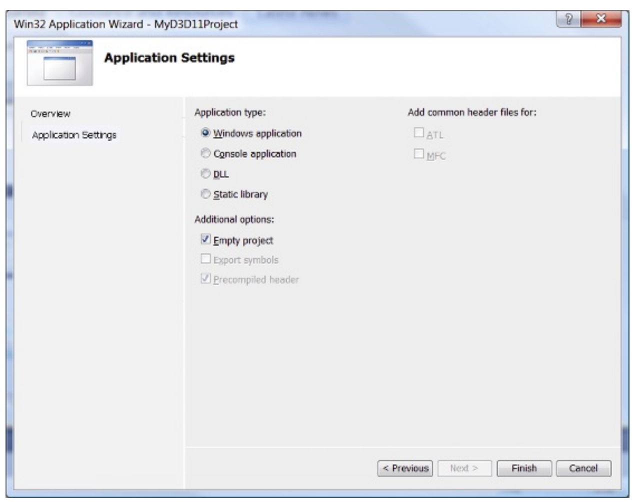


图5　应用程序的相关设置

链接DirectX库

通过在源代码文件Common/d3dApp.h中使用`#pragma`预处理指令来

链接所需的库文件，如：

```
// 链接所需的d3d12库
#pragma comment(lib, "d3dcompiler.lib")
#pragma comment(lib, "D3D12.lib")
#pragma comment(lib, "dxgi.lib")
```

对于创建演示程序而言，该预处理指令使我们免于打开项目属性页面并在连接器配置项下指定附加依赖库。

添加源代码并构建项目

至此，项目已经配置完成。现在来为它添加源代码并对其进行构建。首先，将“Box”演示程序的源代码BoxApp.cpp以及Shaders文件夹（位于d3d12book\Chapter 6 Drawing in Direct3D\Box）复制到工程目录之中。

待复制完上述文件之后，我们以下列步骤来将源代码添加到当前的项目之中。

1．右键单击解决方案资源管理器下的项目名称，在弹出的下拉菜单中依次选择**Add**（添加）→**Existing Item**（现有项），将文件BoxApp.cpp添加到项目中。

2．右键单击解决方案资源管理器下的项目名称，在弹出的下拉菜单中逐步选择**Add**→**Existing Item**，前往读者放置本书**Common**文件夹的位置，并将此文件夹中所有的.h/.cpp文件都添加到项目之中。现在，

方案资源管理器看起来应当与图6相同。

3．再次右键单击解决方案资源管理器下的项目名称，从菜单中选择**Properties**（属性）。再从**Configuration Properties**（配置属性）→**General**（常规）选项卡下，将**Target Platform Version**（目标平台版本）设置为版本**10.x**[4]，以令目标平台为Windows 10。接着单击**Apply**（应用）按钮。

4．大功告成！源代码文件现都已位于项目之中，读者可以在主菜单中选择**Debug**（调试）→**Start Debugging**（开始调试）进行编译、链接以及执行该演示程序。应用程序的执行效果应当与图7所示的一致。

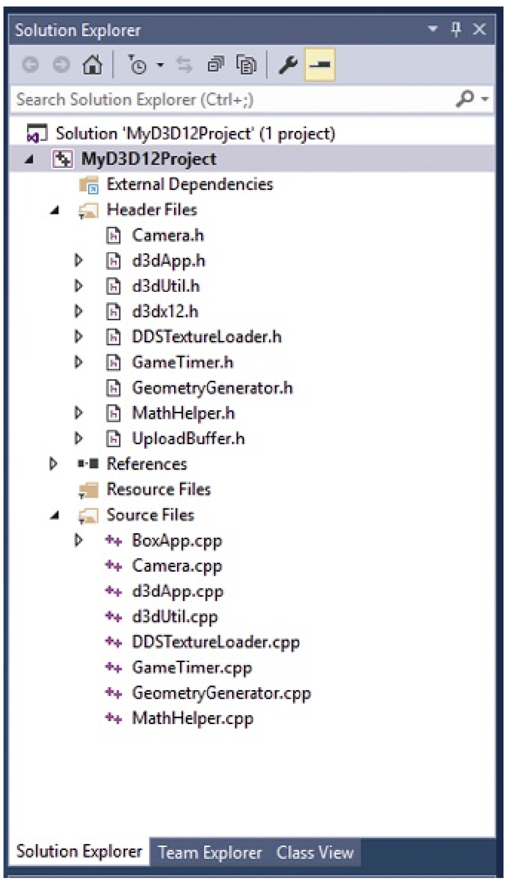


图6　添加“Box”例程所需源代码之后的解决方案资源管理器

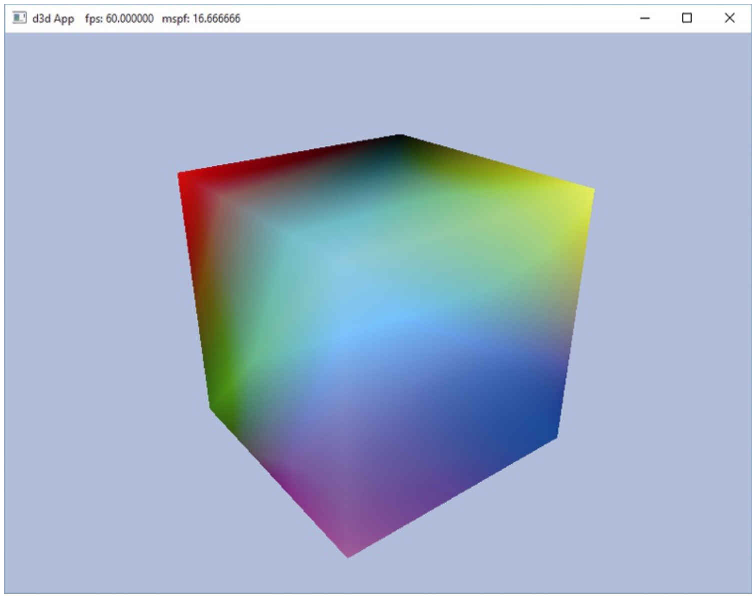


图7 “Box”演示程序的效果

注意

Common目录下的大量代码都是构建本书例程的基石。所以，建议读者先不必忙于查看这些代码。待读到了本书中与之相关的章节后，再研究它们也不迟。

---

[1]　尤其是到了Direct3D 12，更像Mantle等API那样实现了前所未有的更底层的硬件抽象，削减驱动层的工作，转交给开发者负责，从而令图形的处理流程更加“智能”，使用起来犹如贴地飞行的“快感”。

[2]　DirectX包罗系列与多媒体以及游戏开发有关的API，因此Direct3D只是DirectX的一个子集。详细信息请见《DirectX Graphics and Graming》（ee663274）。本书则侧重Direct3D的讲解。

[3]　采用Visual Studio 2017的读者可以参考《Visual Studio中的使用C++的DirectX游戏开发》一文。

[4]　其中的“x”对应于构建项目时所采用的具体SDK版本。

资源与支持

本书由异步社区出品，社区（https://www.epubit.com/）为您提供相关资源和后续服务。

配套资源

本书提供如下资源：

- 本书配套源代码；
- 练习媒体素材；
- 部分练习题解答；
- 书中图片资源。

要获得以上配套资源，请在异步社区本书页面中点击配套资源，跳转到下载界面，按提示进行操作即可。注意：为保证购书读者的权益，该操作会给出相关提示，要求输入提取码进行验证。

如果您是教师，希望获得教学配套资源，请在社区本书页面中直接联系本书的责任编辑。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，点击“提交勘误”，输入勘误信息，点击“提交”按钮即可。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，您将获赠异步社区的100积分。积分可用于在异步社区兑换优惠券、样书或奖品。



与我们联系

我们的联系邮箱是contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们；有意出版图书的作者也可以到异步社区在线提交投稿（直接访问www.epubit.com/selfpublish/submission即可）。

如果您是学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“异步社区”是人民邮电出版社旗下IT专业图书社区，致力于出版精品IT技术图书和相关学习产品，为作译者提供优质出版服务。异步社区创办于2015年8月，提供大量精品IT技术图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网https://www.epubit.com。

“异步图书”是由异步社区编辑团队策划出版的精品IT专业图书的品牌，依托于人民邮电出版社近30年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的LOGO。异步图书的出版领域包括软件开发、大数据、AI、测试、前端、网络技术等。

异步社区

微信服务号

致　　谢

在此，我要对审阅本书早期版本的Rod Lopez、Jim Leiterman、Hanley Leung、Rick Falck、Tybon Wu、Tuomas Sandroos、Eric Sandegren、Jay Tennant与William Goschnick表示感谢。向现存于本书网站上、运用于演示程序中的3D模型以及纹理的制作者Tyler Drinkard致以谢意。我还要感谢Dale E. La Force、Adam Hoult、Gary Simmons、James Lambers以及William Chin，他们曾给予我极大的帮助。另外，亦感激为我提供了DirectX 12 beta版本的Matt Sandy，以及耐心解答用户们在使用beta版时遇到的种种问题的DirectX团队。最后，我还要感谢Mercury Learning and Information出版社的全体工作人员，尤其是出版人David Pallai以及项目经理Jennifer Blaney，是他们直接促成了本书顺利的出版发行。

第一部分　必备的数学知识

“世上之事，无数学则不可解。”

罗杰•培根（《大著作》第四部分，第一章《第一个区别》，1267年）

电子游戏试图向玩家呈现出一个虚拟的世界。然而，计算机从本质上来讲却是一种处理数据的精密仪器。那么问题来了：如何用计算机来表达游戏中虚拟的场景呢？解决的办法就是完全运用数学的方式来描述场景空间以及其中物体的交互。因此，数学在电子游戏的开发中起着至关重要的基础性作用。

在讲述必备知识的第一部分中，我们将介绍穿插于全书的数学工具。重点是向量(vector，物理学和工程学中亦常译为“矢量”)、坐标系（coordinate system）、矩阵（matrix）及其变换（transformation），这些工具将广泛用于本书的所有例程之中。除了对这些数学知识进行讲解以外，我们还将纵览由DirectX数学库所提供的相关类与函数，并示范它们的用法。

请注意，这些主题仅论述了本书后续需要掌握的一些基础内容，而有关电子游戏所需的数学知识却不止于此。对于期望学习更多与游戏相关数学知识的读者，我们推荐[Verth04]和[Lengyel02]。

第**1**章“向量代数”向量（vector）也许是计算机游戏中最基础的数学

对象，没有之一了。例如，我们可以用向量表示位置、位移、方向、速度与力。在这一章中，我们将学习向量及其运算法则。

第**2**章**“**矩阵代数**”**矩阵（matrix）为变换提供了一种高效且紧凑的简化表达方式。在这一章中，我们将熟悉矩阵及其运算定义。

第**3**章**“**变换**”**这一章将考察缩放、旋转和平移这三种基本的几何变换。我们利用这些变换来操纵空间中的3D物体。另外，我们还将讲解坐标变换，以此在不同的坐标系之间转换几何体的坐标表示。

第1章　向量代数

向量在计算机图形学、碰撞检测和物理模拟中扮演着关键的角色，而这几方面又正是构成现代电子游戏的常见组成部分。本书的讲述风格主要趋于实践而非严格化的数学推理，如需要查阅专业的3D游戏或3D图形学数学书籍，可参考[Verth04]一书。需要强调的是，本章研究向量的主要目的在于使读者理解本书中所有例程里向量的用法。

学习目标：

1．学习向量在几何学和数学中的表示方法。

2．了解向量的运算定义及其在几何学中的应用。

3．熟悉DirectXMath库中与向量有关的类和方法。

1.1 向量

向量（vector）是一种兼具大小（也称为模，magnitude）和方向的量。具有这两种属性的量皆称为向量值物理量（vector-valued quantity）。与向量值物理量相关的例子有作用力（在特定方向上施加的力——力的大小即为向量的模）、位移（质点沿净方向[1]移动的距离）和速度（速率和方向）。这样一来，向量就能用于表示力、位移和速度。另外，有时也用向量单指方向，例如玩家在3D游戏里的视角方向、一个多边形的朝向、一束光线的传播方向以及它照射在某表面后的反射方向等。

首先用几何方法来描述向量的数学特征：通过图像中的一条有向线段即可表示一个向量（见图1.1），其中，线段长度代表向量的模，箭头的指向代表向量的方向。我们可以注意到：向量的绘制位置之于其自身是无足轻重的，因为改变某向量的位置并不会对其大小或方向这两个属性造成任何影响。因此，我们说：两个向量相等，当且仅当它们的长度相等且方向相同。所以，图1.1a中的向量$\boldsymbol{u}$和向量$\boldsymbol{v}$相等，因为它们的长度相等且方向相同。事实上，由于位置对于向量是无关紧要的，所以我们总是能在平移一个向量的同时又完全不改变它的几何意义（因为平移操作既不影响它的长度，也不改变它的方向）。显而易见，我们可以将向量$\boldsymbol{u}$完全平移到向量$\boldsymbol{v}$处（反之亦可），使两者完全重合，分毫不差——由此即可证明它们是相等的。现给出一个实例，图1.1b中的向量$\boldsymbol{u}$和向量$\boldsymbol{v}$向两只蚂蚁分别发出指示：令它们从各自所处的两个不同点，A

点和B点，向北爬行10米。这样一来，我们就能根据蚂蚁的爬行路线，再次得到两个相等的向量$\boldsymbol{u}=\boldsymbol{v}$。此时，这两个向量与位置信息无关，仅简单地指挥蚂蚁们如何从它们所处的位置爬行移动。在本例中，蚂蚁们被指示向北（方向）移动10米（长度）。

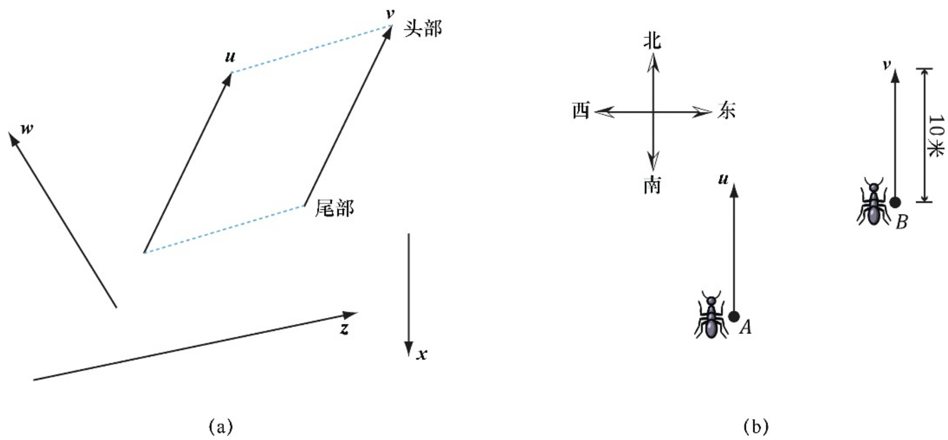


图1.1　向量的实例

（a）绘制在2D平面上的向量　（b）这两个向量指挥蚂蚁们向北移动10米

1.1.1　向量与坐标系

现在来定义向量实用的几何运算，它能解决与向量值物理量有关的问题。然而，由于计算机无法直接处理以几何方法表示的向量，所以需要寻求一种用数学表示向量的方法加以代替。在这里，我们引入一种3D空间坐标系，通过平移操作使向量的尾部都位于原点（见图1.2）。接着，我们就能凭借向量头部的坐标来确定该向量，并将它记作$\boldsymbol{v}=(x,y,z)$，如图1.3所示。现在就能以计算机程序中的3个浮点数来表示一个向量了。

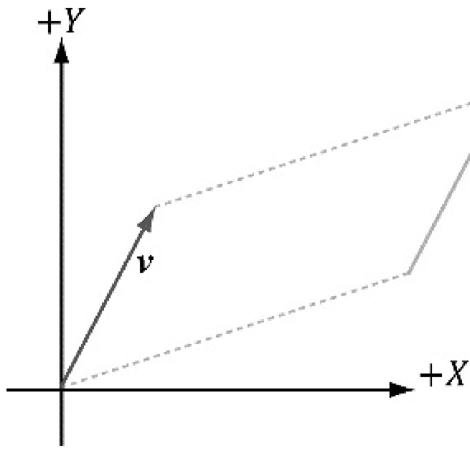


图1.2　平移向量$\boldsymbol{v}$，使它的尾部与坐标系的原点重合。当一个向量的尾部位于原点时，称该向量位于标准位置（standard position）

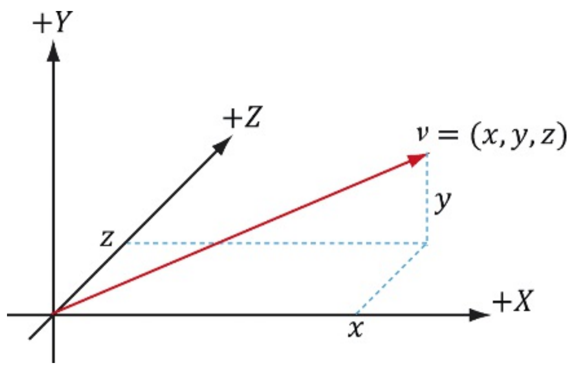


图1.3　一个向量在某3D坐标系中的坐标

注意　Note

如果在2D空间里进行开发工作，则改用2D坐标系即可。此时，向量只有两个坐标分量：$\boldsymbol{v}=(x,y)$。在这种情况下，计算机程序中仅用两个浮点数就能表示一个向量。

请考虑图1.4，该图展示了向量$\boldsymbol{v}$以及空间中两组不同的标架（frame）[2]。我们可以平移向量$\boldsymbol{v}$，将它分别置于两组标架中的标准位置。显而易见的是，向量$\boldsymbol{v}$在标架A中的坐标与它在标架B中的坐标是不同的。换句话说，同一个向量$\boldsymbol{v}$在不同的坐标系中有着不同的坐标表示。

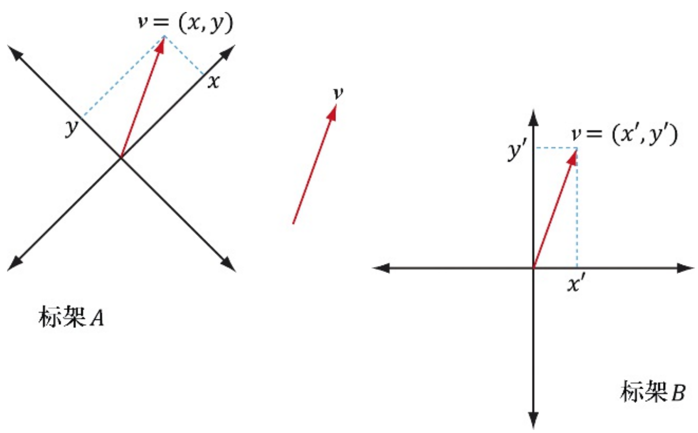


图1.4　同一向量$\boldsymbol{v}$在不同的标架中有着不同的坐标

与此类似的还有温度。水的沸点为100℃或212°F（华氏度）[3]。沸水的物理温度是不变的，与温标无关（也就是说，不能因为采用不同的温标而使其沸点降低），但是我们却可以根据所用的温标来为同一温度赋予不同的标量值。类似地，对于向量来说，它的方向和模都表现在对应的有向线段上，不会更改；只有在改变描述它的参考系时，其坐标才会相应地改变。这一点是很重要的，因为这意味着：每当我们根据坐标来确定一个向量时，其对应的坐标总是相对于某一参考系而言的。在3D计算机图形学中，我们通常会用到较多的参考系。因此，我们需要记录向量在每一种坐标系中的对应坐标。另外，我们也需要知道如何将向量坐标在不同的标架之间进行转换。

注意 Note

可以看出，标架中的向量和点都能够用坐标(x, y, z)来表示。但是它们的意义却是截然不同的：在3D空间中，点仅表示位置，而向量却表示着大小与方向。我们将在1.5节中对点展开进一步的讨论。

1.1.2　左手坐标系与右手坐标系

Direct3D采用的是左手坐标系（left-handed coordinate system）。如果我们伸出左手，并拢手指，假设它们指向的是x轴的正方向，再弯曲四指指向y轴的正方向，则最后伸直拇指的方向大约就是z轴的正方向[4]。图1.5详细展示了左手坐标系与右手坐标系（right-handed coordinate system）的区别。

现在来看右手坐标系。如果伸出右手，并拢手指，假设它们指向的是x轴的正方向，再弯曲四指指向y轴的正方向，那么，最后伸直拇指的方向大约就是z轴的正方向。

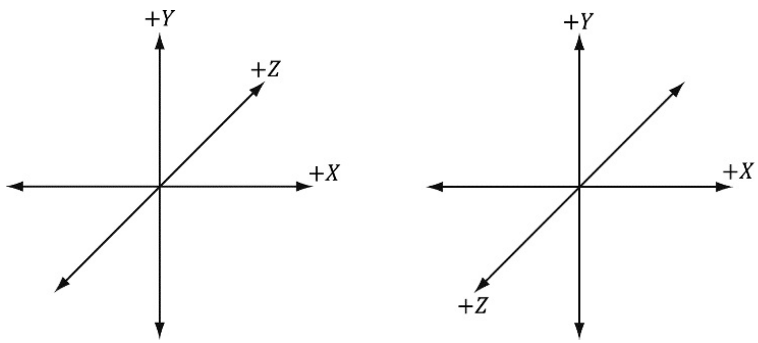


图1.5　图的左侧展示的是左手坐标系，可以看出其中的z坐标轴正方向指向本书页面内；图的右侧展示的是右手坐标系，其z坐标轴正方向则指向页面外

1.1.3　向量的基本运算

现在通过坐标来表示向量的相等、加法运算、标量乘法运算和减法运算的定义。对于这4种定义，设有向量$\boldsymbol{u}=(u_x,u_y,u_z)$和向量$\boldsymbol{v}=(v_x,v_y,v_z)$。

1．两个向量相等，当且仅当它们的对应分量分别相等。即$\boldsymbol{u}=\boldsymbol{v}$，当且仅当$u_x=v_x$，$u_y=v_y$，$u_z=v_z$。

2．向量的加法即令两个向量的对应分量分别相加：$\boldsymbol{u}+\boldsymbol{v}=(u_x+v_x,u_y+v_y,u_z+v_z)$。注意，只有同维的向量之间才可以进行加法运算。

3．向量可以与标量（即实数）相乘，所得到的结果仍是一个向量。例如，设k为一个标量，则$k\boldsymbol{u}=(ku_x,ku_y,ku_z)$。这种运算叫作标量乘法（scalar multiplication）。

4．向量减法可以通过向量加法和标量乘法表示，即

$\boldsymbol{u}-\boldsymbol{v}=\boldsymbol{u}+(-1\cdot\boldsymbol{v})=\boldsymbol{u}+(-\boldsymbol{v})=(u_x-v_x,u_y-v_y,u_z-v_z)$。

例1.1

设向量$\boldsymbol{u}=(1,2,3)$，$\boldsymbol{v}=(1,2,3)$，$\boldsymbol{w}=(3,0,-2)$及标量$k=2$。那么，

1．$\boldsymbol{u}+\boldsymbol{w}=(1,2,3)+(3,0,-2)=(4,2,1)$；

2．$\boldsymbol{u}=\boldsymbol{v}$；

3．$\boldsymbol{u}-\boldsymbol{v}=\boldsymbol{u}+(-\boldsymbol{v})=(1,2,3)+(-1,-2,-3)=(0,0,0)=\mathbf{0}$；

4．$k\boldsymbol{w}=2(3,0,-2)=(6,0,-4)$

第三组运算的不同之处在于其中有个叫作零向量（zero-vector）的特殊向量，它的所有分量都为0，可直接将它简记作$\mathbf{0}$。

例1.2

为了使配图绘制起来更为方便，我们在此例中将围绕2D向量进行讨论。其计算方式与3D向量的方法一致，只不过2D向量少了一个分量而已。

1．设向量$\boldsymbol{v}=(2,1)$，那么该如何在几何学的角度上对$\boldsymbol{v}$与$-\frac{1}{2}\boldsymbol{v}$进行比较呢？我们注意到，$-\frac{1}{2}\boldsymbol{v}=(-1,-\frac{1}{2})$。绘出向量$\boldsymbol{v}$和$-\frac{1}{2}\boldsymbol{v}$（见图1.6a），可以观察到，向量$-\frac{1}{2}\boldsymbol{v}$的方向与向量$\boldsymbol{v}$正好相反，并且长度是向

量$\boldsymbol{v}$的1/2。由此可知，把一个向量的系数变为其相反数，就相当于在几何学中“翻转”此向量的方向，而且对向量进行标量乘法即为对其长度进行缩放。

2．设向量$\boldsymbol{u}=(2,\frac{1}{2})$，$\boldsymbol{v}=(1,2)$，则$\boldsymbol{u}+\boldsymbol{v}=(3,\frac{5}{2})$。图1.6b展示了向量加法运算的几何意义：把向量$\boldsymbol{u}$进行平移，使$\boldsymbol{u}$的尾部与$\boldsymbol{v}$的头部重合。此时，向量$\boldsymbol{u}$与向量$\boldsymbol{v}$的和即：以$\boldsymbol{v}$的尾部为起点、以平移后$\boldsymbol{u}$的头部为终点所作的向量（如果令向量$\boldsymbol{u}$的位置保持不变，平移向量$\boldsymbol{v}$，使$\boldsymbol{v}$的尾部与$\boldsymbol{u}$的头部重合也能得到同样的结果。在这种情况下，$\boldsymbol{u}+\boldsymbol{v}$的和就可以表示为以$\boldsymbol{u}$的尾部为起点、以平移后$\boldsymbol{v}$的头部为终点所作的向量）。可以看出，向量的加法运算与物理学中不同作用力合成合力的规则是一致的。如果有两个力（两个向量）作用在同一方向上，则将在这个方向上产生更大的合力（更长的向量）；如果有两个力（两个向量）作用于彼此相反的方向上，那么便会产生更小的合力（更短的向量），如图1.7所示。

3．设向量$\boldsymbol{u}=(2,\frac{1}{2})$，$\boldsymbol{v}=(1,2)$，则$\boldsymbol{v}-\boldsymbol{u}=(-1,\frac{3}{2})$。图1.6c展示了向量减法运算的几何意义。从本质上讲，$\boldsymbol{v}-\boldsymbol{u}$的差值仍是一个向量，该向量自$\boldsymbol{u}$的头部始至$\boldsymbol{v}$的头部终。如果我们将$\boldsymbol{u}$和$\boldsymbol{v}$看作两个点，那么$\boldsymbol{v}-\boldsymbol{u}$得到的是一个从点$\boldsymbol{u}$指向点$\boldsymbol{v}$的向量；这种解释方式的重点在于使我们找出向量的方向。同时，不难看出，在把$\boldsymbol{u}$与$\boldsymbol{v}$看作点的时候，$\boldsymbol{v}-\boldsymbol{u}$的长度也就是“点$\boldsymbol{u}$到点$\boldsymbol{v}$的距离”。

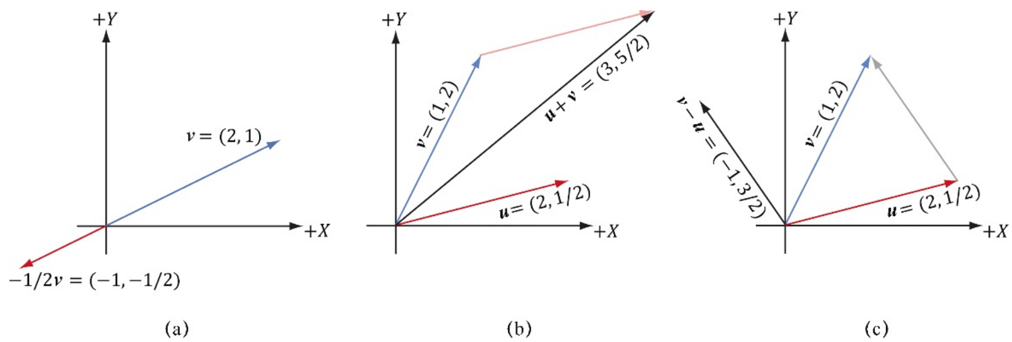


图1.6　向量运算的几何意义

（a）标量乘法的几何意义　（b）向量加法的几何意义　（c）向量减法的几何意义

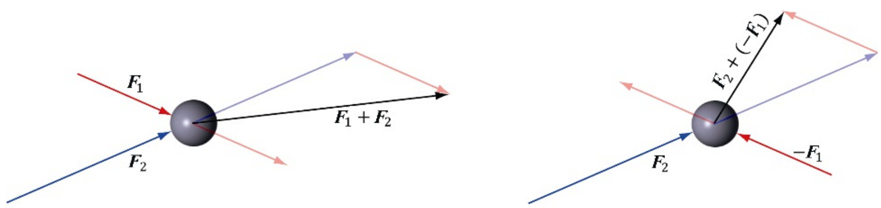


图1.7　作用在球上的两个作用力。利用向量加法将两者合成为一个合力

1.2　长度和单位向量

向量大小（亦称为模）的几何意义是对应有向线段的长度，用双竖线表示（例如$\|\boldsymbol{u}\|$代表向量$\boldsymbol{u}$的模）。现给出向量$\boldsymbol{u}=(x,y,z)$，我们希望用代数的方法计算它的模。3D向量的模可通过运用两次毕达哥拉斯定理[5]得出，如图1.8所示。

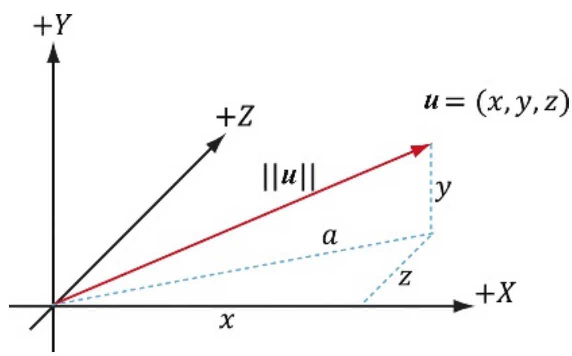


图1.8　运用两次毕达哥拉斯定理便能得出3D向量的模

首先来看位于平面xz中以x，z为直角边，以a为斜边所构成的直角三角形。根据毕达哥拉斯定理，有$a=\sqrt{x^2+z^2}$。接下来再看以a，y为直角边，以$\|\boldsymbol{u}\|$为斜边所围成的直角三角形。再次运用毕达哥拉斯定理，便能得出下列计算向量模的公式：

$$\|\boldsymbol{u}\|=\sqrt{y^2+a^2}=\sqrt{y^2+\left(\sqrt{x^2+z^2}\right)^2}=\sqrt{x^2+y^2+z^2} \tag{1.1}$$

在某些情况下，我们并不关心向量的长度，仅用它来表示方向。对此，我们希望使该向量的长度为1。把一个向量的长度变为单位长度称为向量的**规范化**[6]（normalizing）处理。具体实现方法是，将向量的每

个分量分别除以该向量的模：

$$\hat{\boldsymbol{u}} = \frac{\boldsymbol{u}}{||\boldsymbol{u}||} = \left(\frac{x}{||\boldsymbol{u}||}, \frac{y}{||\boldsymbol{u}||}, \frac{z}{||\boldsymbol{u}||}\right) \qquad (1.2)$$

为了验证公式的正确性，下面计算$\hat{\boldsymbol{u}}$的长度：

$$||\hat{\boldsymbol{u}}|| = \sqrt{\left(\frac{x}{||\boldsymbol{u}||}\right)^2 + \left(\frac{y}{||\boldsymbol{u}||}\right)^2 + \left(\frac{z}{||\boldsymbol{u}||}\right)^2} = \frac{\sqrt{x^2+y^2+z^2}}{\sqrt{||\boldsymbol{u}||^2}} = \frac{||\boldsymbol{u}||}{||\boldsymbol{u}||} = 1$$

由此可见，$\hat{\boldsymbol{u}}$确实是一个单位向量（unit vector）。

例1. 3

对向量$\boldsymbol{v} = (-1, 3, 4)$进行规范化处理。我们能求出$||\boldsymbol{v}|| = \sqrt{(-1)^2 + 3^2 + 4^2} = \sqrt{26}$。因此，

$$||\boldsymbol{v}|| = \frac{\boldsymbol{v}}{||\boldsymbol{v}||} = \left(-\frac{1}{\sqrt{26}}, \frac{3}{\sqrt{26}}, \frac{4}{\sqrt{26}}\right)$$

为了验证$\hat{\boldsymbol{v}}$是单位向量，我们计算其长度：

$$||\hat{\boldsymbol{v}}|| = \sqrt{\left(-\frac{1}{\sqrt{26}}\right)^2 + \left(\frac{3}{\sqrt{26}}\right)^2 + \left(\frac{4}{\sqrt{26}}\right)^2} = \sqrt{\frac{1}{26} + \frac{9}{26} + \frac{16}{26}} = \sqrt{1} = 1$$

1.3　点积

点积（dot product，亦称数量积或内积）是一种计算结果为标量值的向量乘法运算，因此有时也称为标量积（scalar product）。设向量$\boldsymbol{u}=(u_x,u_y,u_z)$，$\boldsymbol{v}=(v_x,v_y,v_z)$，则点积的定义为：

$$\boldsymbol{u}\cdot\boldsymbol{v}=u_xv_x+u_yv_y+u_zv_z \qquad (1.3)$$

可见，点积就是向量间对应分量的乘积之和。

点积的定义并没有明显地体现出其几何意义。但是我们却能根据余弦定理（law of cosines，参见练习10）找到二向量点积的几何关系：

$$\boldsymbol{u}\cdot\boldsymbol{v}=\|\boldsymbol{u}\|\ \|\boldsymbol{v}\|\cos\theta \qquad (1.4)$$

其中，θ是向量$\boldsymbol{u}$与向量$\boldsymbol{v}$之间的夹角，$0\leqslant\theta\leqslant\pi$，如图1.9所示。式（1.4）表明，两向量的点积为：两向量夹角的余弦值乘以这两个向量的模。特别地，如果向量$\boldsymbol{u}$和向量$\boldsymbol{v}$都是单位向量，那么$\boldsymbol{u}\cdot\boldsymbol{v}$就等于两向量夹角的余弦值，即$\boldsymbol{u}\cdot\boldsymbol{v}=\cos\theta$。

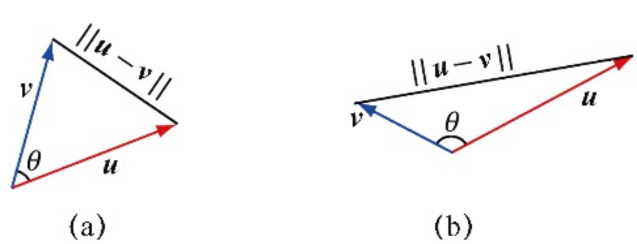


图1.9　图a中，向量$\boldsymbol{u}$与向量$\boldsymbol{v}$之间的夹角θ是一个锐角；图b中，向量$\boldsymbol{u}$与向量$\boldsymbol{v}$之间的夹角θ是一个钝角。每当讨论两个向量之间的夹角时，我们提及的总是较小的那个角，即角θ总是满足$0\leqslant\theta\leqslant\pi$

式（1.4）给出了一些有用的点积几何性质：

1．如果$\boldsymbol{u}\cdot\boldsymbol{v}=0$，那么$\boldsymbol{u}\perp\boldsymbol{v}$（即两个向量正交）。

2．如果$\boldsymbol{u}\cdot\boldsymbol{v}>0$，那么两向量之间的夹角$\theta$小于90°（即两向量间的夹角为一锐角）。

3．如果$\boldsymbol{u}\cdot\boldsymbol{v}<0$，那么两向量之间的夹角$\theta$大于90°（即两向量间的夹角为一钝角）。

注意 Note

“正交”（orthogonal）与“垂直”（perpendicular）实为同义词。

例1.4

设向量$\boldsymbol{u}=(1,2,3)$，$\boldsymbol{v}=(-4,0,-1)$。计算向量$\boldsymbol{u}$与$\boldsymbol{v}$之间的夹角。

先来计算：

$$\boldsymbol{u}\cdot\boldsymbol{v}=(1,\ 2,\ 3)\cdot(-4,\ 0,\ -1)=-4-3=-7$$

$$\|\boldsymbol{u}\|=\sqrt{1^2+2^2+3^2}=\sqrt{14}$$

$$\|\boldsymbol{v}\|=\sqrt{(-4)^2+0^2+(-1)^2}=\sqrt{17}$$

现在，运用式（1.4）得到θ：

$$\cos\theta = \frac{\boldsymbol{u}\cdot\boldsymbol{v}}{\|\boldsymbol{u}\|\ \|\boldsymbol{v}\|} = \frac{-7}{\sqrt{14}\cdot\sqrt{17}}$$

$$\theta = \cos^{-1}\frac{-7}{\sqrt{14}\cdot\sqrt{17}} \approx 117^\circ$$

例1.5

考虑图1.10。给出向量$\boldsymbol{v}$和单位向量$\boldsymbol{n}$，请借助点积公式求出用$\boldsymbol{v}$和$\boldsymbol{n}$表示向量$\boldsymbol{p}$的公式。

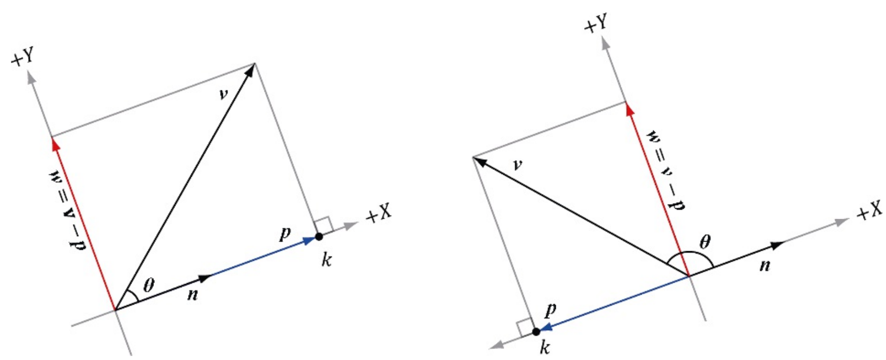


图1.10　向量$\boldsymbol{v}$在单位向量$\boldsymbol{n}$上的正交投影（orthogonal projection）

首先，观察图示可以得知存在一标量k，使得$\boldsymbol{p} = k\boldsymbol{n}$；而且，因为我们假设$\|\boldsymbol{n}\| = 1$，所以有$\|\boldsymbol{p}\| = \|k\boldsymbol{n}\| = \|k\ \|\|\ \boldsymbol{n}\| = \|k\|$。注意，$k$可能是负值，当且仅当$\boldsymbol{p}$与$\boldsymbol{n}$的方向相反。利用三角函数，我们有$k = \|\boldsymbol{v}\|\cos\theta$；因此，$\boldsymbol{p} = k\boldsymbol{n} = (\|\boldsymbol{v}\|\cos\theta)\boldsymbol{n}$。又由于$\boldsymbol{n}$是单位向量，便可以用另一种方法来表示：

$$\boldsymbol{p} = (\|\boldsymbol{v}\|\cos\theta)\boldsymbol{n} = (\|\boldsymbol{v}\|\cdot 1\cos\theta)\boldsymbol{n} = (\|\boldsymbol{v}\|\ \|\boldsymbol{n}\|\cos\theta)\boldsymbol{n} = (\boldsymbol{v}\cdot\boldsymbol{n})\boldsymbol{n}$$

特别是这里证明了：当$\boldsymbol{n}$是单位向量时，$k = \boldsymbol{v}\cdot\boldsymbol{n}$，顺带也解释了在

这种情况下$\boldsymbol{v}\cdot\boldsymbol{n}$的几何意义。我们称$\boldsymbol{p}$为向量$\boldsymbol{v}$落在向量$\boldsymbol{n}$上的正交投影（orthogonal projection），通常将它表示为：

$$\boldsymbol{p}=\mathrm{proj}_{\boldsymbol{n}}(\boldsymbol{v})$$

如果将$\boldsymbol{v}$看作是一个力，便可认为$\boldsymbol{p}$是力$\boldsymbol{v}$在方向$\boldsymbol{n}$上的分力。同理，向量$\boldsymbol{w}=\mathrm{perp}_{\boldsymbol{n}}(\boldsymbol{v})=\boldsymbol{v}-\boldsymbol{p}$是作用力$\boldsymbol{v}$在$\boldsymbol{n}$的正交方向上的分力（这就是用$\mathrm{perp}_{\boldsymbol{n}}(\boldsymbol{v})$来表示“垂直”的原因）。观察到$\boldsymbol{v}=\boldsymbol{p}+\boldsymbol{w}=\mathrm{proj}_{\boldsymbol{n}}(\boldsymbol{v})+\mathrm{perp}_{\boldsymbol{n}}(\boldsymbol{v})$，这就是说，可以将向量$\boldsymbol{v}$分解成两个互相正交的向量$\boldsymbol{p}$与$\boldsymbol{w}$之和。

如果$\boldsymbol{n}$不具有单位长度，就先对它进行规范化处理，使之成为单位向量。通过把向量$\boldsymbol{n}$替换为单位向量$\frac{\boldsymbol{n}}{\|\boldsymbol{n}\|}$，即可得到更具一般性的投影公式：

$$\boldsymbol{p}=\mathrm{proj}_{\boldsymbol{n}}(\boldsymbol{v})=\left(\boldsymbol{v}\cdot\frac{\boldsymbol{n}}{\|\boldsymbol{n}\|}\right)\frac{\boldsymbol{n}}{\|\boldsymbol{n}\|}=\frac{(\boldsymbol{v}\cdot\boldsymbol{n})}{\|\boldsymbol{n}\|^2}\boldsymbol{n}$$

正交化

如果向量集$\{\boldsymbol{v}_0,\cdots,\boldsymbol{v}_{n-1}\}$中的每个向量都是互相正交（集合内的任一向量都与集合中的其他所有向量相互正交）且皆具单位长度，那么我们就称此集合是规范正交（orthonormal）的。有时我们会接到一个近乎（但并不完全）规范正交的集合。这时，一个常见的工作就是通过正交化手段，使之成为规范正交集。例如，我们有时会在3D计算机图形学中用到规范正交集，但是由于处理过程中数值精度的问题，它会随之逐

步变为非规范正交集。这时就要用到正交化这一手段了。我们下面将主要围绕这种问题的2D和3D情况展开探讨（也就是说，集合内只有2个或3个向量的情况）。

先来考察相对简单的2D情况吧。假设我们有向量集合$\{\boldsymbol{v}_0, \boldsymbol{v}_1\}$，现欲将它正交化为图1.11中所示的正交集$\{\boldsymbol{w}_0, \boldsymbol{w}_1\}$。首先设$\boldsymbol{w}_0 = \boldsymbol{v}_0$，通过使$\boldsymbol{v}_1$减去它在$\boldsymbol{w}_0$上的分量（投影）来令它正交于$\boldsymbol{w}_0$：

$$\boldsymbol{w}_1 = \boldsymbol{v}_1 - \mathrm{proj}_{\boldsymbol{w}0}(\boldsymbol{v}_1)$$

此时，我们便得到了一个元素互相正交的向量集合$\{\boldsymbol{w}_0, \boldsymbol{w}_1\}$；最后一步是构建一个规范正交集，将向量$\boldsymbol{w}_0$和$\boldsymbol{w}_1$规范化为单位向量即可。

3D情况与2D情况的处理方法相似，但是步骤更多。假设有向量集$\{\boldsymbol{v}_0, \boldsymbol{v}_1, \boldsymbol{v}_2\}$，现希望将它正交化为正交集$\{\boldsymbol{w}_0, \boldsymbol{w}_1, \boldsymbol{w}_2\}$，过程如图1.12所示。首先使$\boldsymbol{w}_0 = \boldsymbol{v}_0$，通过令$\boldsymbol{v}_1$减去它在$\boldsymbol{w}_0$方向上的分量，让它正交于$\boldsymbol{w}_0$：

$$\boldsymbol{w}_1 = \boldsymbol{v}_1 - \mathrm{proj}_{\boldsymbol{w}0}(\boldsymbol{v}_1)$$

接下来，通过令$\boldsymbol{v}_2$依次减去它在$\boldsymbol{w}_0$方向与$\boldsymbol{w}_1$方向上的分量（投影），使之同时正交于$\boldsymbol{w}_0$、$\boldsymbol{w}_1$：

$$\boldsymbol{w}_2 = \boldsymbol{v}_2 - \mathrm{proj}_{\boldsymbol{w}_0}(\boldsymbol{v}_2) - \mathrm{proj}_{\boldsymbol{w}_1}(\boldsymbol{v}_2)$$

现在我们就得到了所有元素都彼此正交的向量集$\{\boldsymbol{w}_0, \boldsymbol{w}_1, \boldsymbol{w}_2\}$；最后一步是通过将$\boldsymbol{w}_0$、$\boldsymbol{w}_1$和$\boldsymbol{w}_2$规范化为单位向量来构建一个规范正交集。

对于具有n个向量的一般集合$\{\boldsymbol{v}_0, \cdots, \boldsymbol{v}_{n-1}\}$而言，为了将其正交化为规范正交集$\{\boldsymbol{w}_0, \cdots, \boldsymbol{w}_{n-1}\}$，我们就要使用格拉姆—施密特正交化（Gram-Schmidt Orthogonalization）方法进行处理。

基本步骤：设$\boldsymbol{w}_0 = \boldsymbol{v}_0$

对于$1 \leqslant i \leqslant n-1$，令$$\boldsymbol{w}_i = \boldsymbol{v}_i - \sum_{j=0}^{i-1} \mathrm{proj}_{\boldsymbol{w}_j}(\boldsymbol{v}_i)$$

规范化步骤：令$$\boldsymbol{w}_i = \frac{\boldsymbol{w}_i}{||\boldsymbol{w}_i||}$$

再次重申，从直观上来说，在将给定集合内的向量$\boldsymbol{v}_i$添加到规范正交集中时，我们需要令$\boldsymbol{v}_i$减去它在现有规范正交集中其他向量$\{\boldsymbol{w}_0, \boldsymbol{w}_1, \cdots, \boldsymbol{w}_{i-1}\}$方向上的分量（投影），这样方可确保新加入规范正交集的向量与该集合中的其他向量互相正交。

1.4　叉积

向量乘法的第二种形式是叉积（cross product，亦称向量积、外积）。与计算结果为标量的点积不同，叉积的计算结果亦为向量。此外，只有3D向量的叉积有定义（不存在2D向量叉积）。假设3D向量$\boldsymbol{u}$和$\boldsymbol{v}$的叉积得到的是另一个向量$\boldsymbol{w}$，则$\boldsymbol{w}$与向量$\boldsymbol{u}$、$\boldsymbol{v}$彼此正交。也就是说，向量$\boldsymbol{w}$既正交于$\boldsymbol{u}$，也正交于$\boldsymbol{v}$，如图1.13所示。如果$\boldsymbol{u} = (u_x, u_y, u_z)$，$\boldsymbol{v} = (v_x, v_y, v_z)$，那么叉积的计算方法为：

$$\boldsymbol{w} = \boldsymbol{u} \times \boldsymbol{v} = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x) \qquad (1.5)$$

注意　Note

若实际采用的是右手坐标系，则遵守右手拇指法则（right-hand-thumb rule，有的文献也称之为右手定则）：如果伸出右手并拢手指，令它们指向第一个向量$\boldsymbol{u}$的方向，再以$0 \leqslant \theta \leqslant \pi$的角度弯曲四指，使之指向向量$\boldsymbol{v}$的方向，那么，最后伸直拇指的方向大约为向量$\boldsymbol{w} = \boldsymbol{u} \times \boldsymbol{v}$的方向。

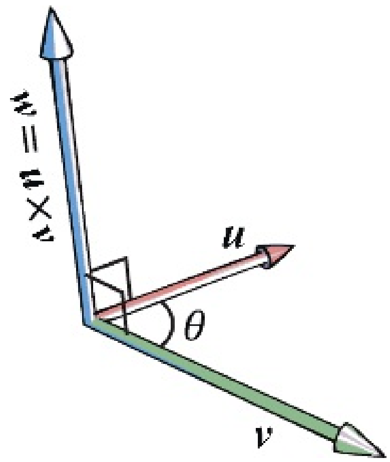


图1.13　两个3D向量$\boldsymbol{u}$与$\boldsymbol{v}$的叉积得到的是：既正交于$\boldsymbol{u}$也正交于$\boldsymbol{v}$的向量$\boldsymbol{w}$。如果伸出左手，使并拢的左手手指指向向量$\boldsymbol{u}$的方向，再以$0 \leqslant \theta \leqslant \pi$的角度弯曲四指，使之指向向量$\boldsymbol{v}$的方向，

那么最后伸直的大拇指约略指向的即为$\boldsymbol{w} = \boldsymbol{u} \times \boldsymbol{v}$的方向。

这就是所谓的左手拇指法则（left-hand-thumb rule，有的文献也称之为左手定则）

例1.6

设向量$\boldsymbol{u} = (2, 1, 3)$和向量$\boldsymbol{v} = (2, 0, 0)$。计算$\boldsymbol{w} = \boldsymbol{u} \times \boldsymbol{v}$与$\boldsymbol{z} = \boldsymbol{v} \times \boldsymbol{u}$，并验证向量$\boldsymbol{w}$既正交于向量$\boldsymbol{u}$又正交于向量$\boldsymbol{v}$。运用式（1.5），有：

$$\begin{aligned}\boldsymbol{w} &= \boldsymbol{u} \times \boldsymbol{v} \\ &= (2,\ 1,\ 3) \times (2,\ 0,\ 0) \\ &= (1 \cdot 0 - 3 \cdot 0,\ 3 \cdot 2 - 2 \cdot 0,\ 2 \cdot 0 - 1 \cdot 2) \\ &= (0,\ 6,\ -2)\end{aligned}$$

以及

$$\begin{aligned}\boldsymbol{z} &= \boldsymbol{v} \times \boldsymbol{u} \\ &= (2,\ 0,\ 0) \times (2,\ 1,\ 3) \\ &= (0 \cdot 3 - 0 \cdot 1,\ 0 \cdot 2 - 2 \cdot 3,\ 2 \cdot 1 - 0 \cdot 2) \\ &= (0,\ -6,\ 2)\end{aligned}$$

根据计算结果可以明确地得出一项结论：一般来说$\boldsymbol{u}\times\boldsymbol{v}\neq\boldsymbol{v}\times\boldsymbol{u}$，即向量的叉积不满足交换律。事实上，我们同时也能够证明$\boldsymbol{u}\times\boldsymbol{v}=-\boldsymbol{v}\times\boldsymbol{u}$，这正是叉积的反交换律。叉积所得的向量可以通过左手拇指法则来加以确认。伸出左手，如果并拢手指指向的为参与叉积运算第一个向量的方向，再弯曲四指指向参与叉积运算第二个向量的方向（总是按两者间较小的夹角弯曲四指。如果无法做到，四指需要向手背方向旋转，则说明手心要转到背对方向，拇指最终指向相反方向），那么伸直的拇指方向即为所求叉积的向量方向，如图1.13所示。

为了证明向量$\boldsymbol{w}$既正交于向量$\boldsymbol{u}$又正交于向量$\boldsymbol{v}$，我们需要用到1.3节中的结论：如果$\boldsymbol{u}\cdot\boldsymbol{v}=0$，那么$\boldsymbol{u}\perp\boldsymbol{v}$（即两个向量彼此正交）。由于：

$$\boldsymbol{w}\cdot\boldsymbol{u}=(0,6,-2)\cdot(2,1,3)=0\cdot 2+6\cdot 1+(-2)\cdot 3=0$$

以及

$$\boldsymbol{w}\cdot\boldsymbol{v}=(0,6,-2)\cdot(2,0,0)=0\cdot 2+6\cdot 0+(-2)\cdot 0=0$$

由此可以推断出：向量$\boldsymbol{w}$既正交于向量$\boldsymbol{u}$，也正交于向量$\boldsymbol{v}$。

1.4.1 2D向量的伪叉积

我们刚刚证明了：通过叉积可以求出与两个指定3D向量正交的向量。在 2D 空间中虽然不存在这种情况，但是若给定一个2D向量$\boldsymbol{u}=(u_x,u_y)$，我们还是能通过与3D向量叉积相似的方法，求出与$\boldsymbol{u}$正交的向量$\boldsymbol{v}$。图1.14从几何角度展示了满足上述条件的向量$\boldsymbol{v}=(-u_y,u_x)$。形式上的证明也比较简洁：

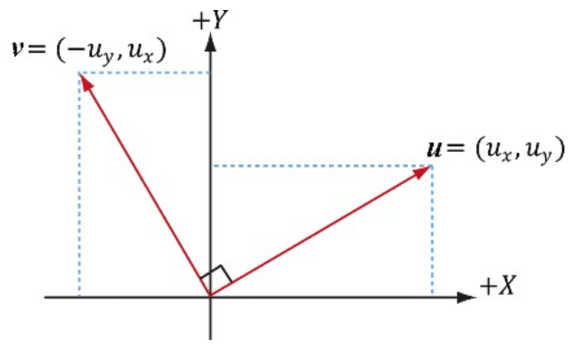


图1.14　向量$\boldsymbol{u}$的2D伪叉积计算结果是正交于$\boldsymbol{u}$的向量$\boldsymbol{v}$

$$\boldsymbol{u} \cdot \boldsymbol{v} = (u_x, u_y) \cdot (-u_y, u_x) = -u_x u_y + u_y u_x = 0$$

因此，$\boldsymbol{u} \perp \boldsymbol{v}$。同时，不难看出$\boldsymbol{u} \cdot -\boldsymbol{v} = u_x u_y + u_y(-u_x) = 0$，所以亦可知$\boldsymbol{u} \perp -\boldsymbol{v}$。

1.4.2　通过叉积来进行正交化处理

在1.3.1节中，我们曾探讨了可以使向量集正交化的方法：格拉姆—施密特正交化方法。对于3D情况来讲，还存在另外一种与叉积有关的策略，可使近乎规范正交的向量集$\{\boldsymbol{v}_0, \boldsymbol{v}_1, \boldsymbol{v}_2\}$完全正交化。但若受数值精度误差累积的影响，也许会导致其成为非规范正交集。图1.15中几何图示所对照的叉积处理流程如下。

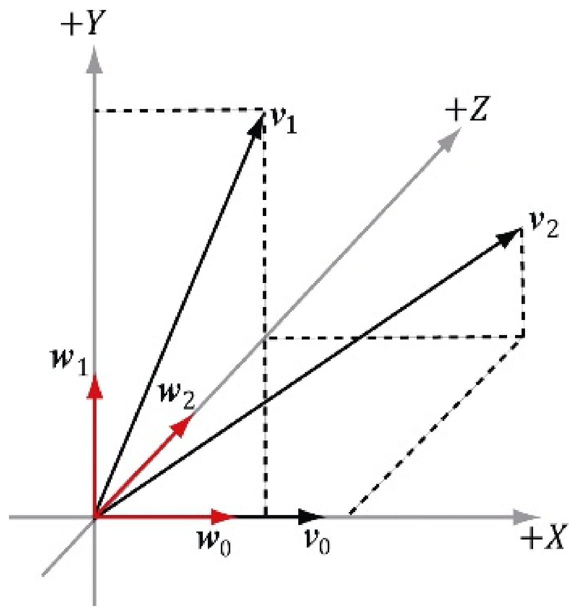


图1.15　通过叉积来进行正交化处理3D正交化处理。

1.　令$\boldsymbol{w}_0 = \frac{\boldsymbol{v}_0}{||\boldsymbol{v}_0||}$。

2．令$\boldsymbol{w}_2 = \frac{\boldsymbol{w}_0 \times \boldsymbol{v}_1}{||\boldsymbol{w}_0 \times \boldsymbol{v}_1||}$。

3．令$\boldsymbol{w}_1 = \boldsymbol{w}_2 \times \boldsymbol{w}_0$，根据练习14可知：由于$\boldsymbol{w}_2 \perp \boldsymbol{w}_0$且$||\boldsymbol{w}_2|| = ||\boldsymbol{w}_0|| = 1$，因此$||\boldsymbol{w}_2 \times \boldsymbol{w}_0|| = 1$。所以，我们最后也就不再需要对它进行规范化处理了。

此时，向量集$\{\boldsymbol{w}_0, \boldsymbol{w}_1, \boldsymbol{w}_2\}$是规范正交的。

注意 Note

在上面的示例中，我们首先令$\boldsymbol{w}_0 = \frac{\boldsymbol{v}_0}{||\boldsymbol{v}_0||}$，这意味着将向量$\boldsymbol{v}_0$转换到

向量$\boldsymbol{w}_0$时并未改变方向——仅缩放了$\boldsymbol{v}_0$的长度而已。但是，向量$\boldsymbol{w}_1$与向量$\boldsymbol{w}_2$的方向却可以分别不同于向量$\boldsymbol{v}_1$和向量$\boldsymbol{v}_2$。对于特定的应用来说，不改变集合中某个向量的方向也许是件很重要的事。例如，在本书后面，我们会利用3个规范正交向量$\{\boldsymbol{v}_0, \boldsymbol{v}_1, \boldsymbol{v}_2\}$来表示摄像机（camera）的朝向，而其中的第三个向量$\boldsymbol{v}_2$描述的正是摄像机的观察方向。在对这些向量进行正交化处理的过程中，我们通常并不希望改变此摄像机的观察方向。所以，我们会运用上面的算法，在第一步中处理向量$\boldsymbol{v}_2$，再通过修改向量$\boldsymbol{v}_0$和向量$\boldsymbol{v}_1$来使它们正交化。

1.5　点

到目前为止，我们一直都在讨论向量，却还没有对位置的概念进行任何描述。然而，在3D程序中是需要我们来指明位置关系的，例如3D几何体的位置和3D虚拟摄像机的位置等。在一个坐标系中，通过一个处于标准位置的向量（见图1.16）就能表示出3D空间中的特定位置，我们称这种向量为位置向量（position vector）。在这种情况下，向量箭头的位置才是值得关注的主要特征，而方向和大小都是无足轻重的。“位置向量”和“点”这两个术语可以互相替代，这是因为一个位置向量足以确定一个点。

然而，用向量表示点也有副作用，在代码中则更为明显，因为部分向量运算对点来说是没有意义的。例如，两点之和的意义何在？但从另一方面来讲，一些运算却可以在点上得到推广。如，可以将两个点的差$\boldsymbol{q}-\boldsymbol{p}$定义为由点$\boldsymbol{p}$指向点$\boldsymbol{q}$的向量。同样，也可以定义点$\boldsymbol{p}$与向量$\boldsymbol{v}$相加，其意义为：令点$\boldsymbol{p}$沿向量$\boldsymbol{v}$位移而得到点$\boldsymbol{q}$。由于我们用向量来表示坐标系中的点，所以除了刚刚讨论过的几类与点有关的运算外便无须再做其他额外的工作，这是因为利用向量代数的框架就足以解决点的描述问题了，详见图1.17。

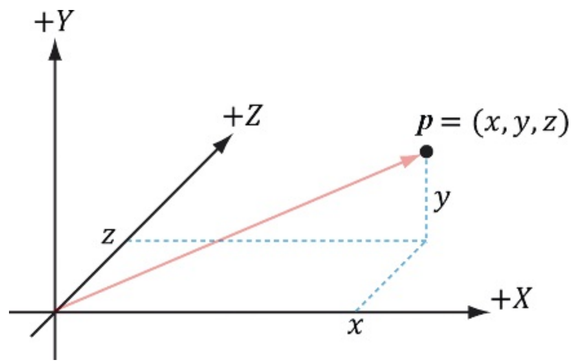


图1.16　由原点延伸至目标点的位置向量，用它即可描述目标点在坐标系中的位置

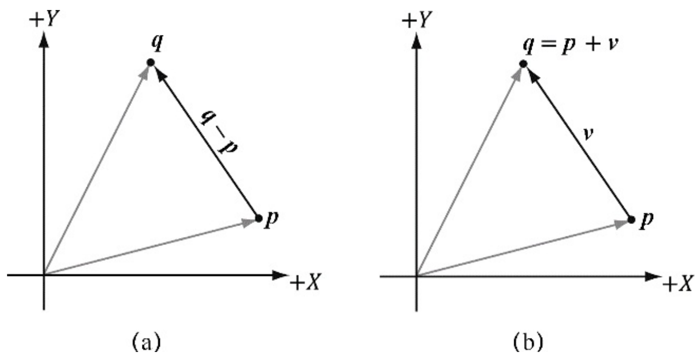


图1.17　图a通过$\mathbf{q}-\mathbf{p}$的两点之差来定义由点$\mathbf{p}$指向点$\mathbf{q}$的向量。图b中点$\mathbf{p}$与向量$\mathbf{v}$的和可以定义为：使点$\mathbf{p}$沿着向量$\mathbf{v}$位移而得到点$\mathbf{q}$

注意

其实还有一种通过几何方式来定义的多点之间的特殊和，即仿射组合（affine combination），这种运算的过程就像求取诸点的加权平均值。

1.6　利用DirectXMath库进行向量运算

对于Windows 8及其以上版本来讲，DirectXMath（其前身为XNA Math数学库，DirectXMath正是基于此而成）是一款为Direct3D应用程序量身打造的3D数学库，而它也自此成为了Windows SDK的一部分。该数学库采用了SIMD流指令扩展2（Streaming SIMD Extensions 2，SSE2）指令集。借助128位宽的单指令多数据（Single Instruction Multiple Data，SIMD）寄存器，利用一条SIMD指令即可同时对4个32位浮点数或整数进行运算。这对于向量运算带来的益处是不言而喻的。例如，若见到如下的向量加法：

$$\boldsymbol{u}+\boldsymbol{v}=(u_x+v_x, u_y+v_y, u_z+v_z)$$

我们按普通的计算方式只能对分量逐个相加。而通过SIMD技术，我们就可以仅用一条SIMD加法指令来取代4条普通的标量指令，从而直接计算出4D向量的加法结果。如果只需要进行3D数据运算，我们仍然可以使用SIMD技术，但是要忽略第4个坐标分量。类似地，对于2D运算，则应忽略第3、4个坐标分量。

我们并不会对DirectXMath库进行全面的介绍，而只是针对本书需要的关键部分进行讲解。关于此库的所有细节，可以参考它的在线文档[DirectXMath]。对于希望了解如何开发一个优秀的SIMD向量库，乃至希望深入理解DirectXMath库设计原理的读者，我们在这里推荐一篇文章《Designing Fast Cross-Platform SIMD Vector Libraries（设计快速的跨平台SIMD向量库）》[Oliveira 2010]。

为了使用DirectXMath库，我们需要向代码中添加头文件`#include <DirectXMath.h>`，而为了一些相关的数据类型还要加入头文件`#include <DirectXPackedVector.h>`。除此之外并不需要其他的库文件，因为所有的代码都以内联的方式实现在头文件里。DirectXMath.h文件中的代码都存在于`DirectX`命名空间之中，而DirectXPackedVector.h文件中的代码则都位于`DirectX::PackedVector`命名空间以内。另外，针对x86平台，我们需要启用SSE2指令集（**Project Properties**（工程属性）→**Configuration Properties**（配置属性）→**C/C++**→**Code Generation**（代码生成）→**Enable Enhanced Instructon Set**（启用增强指令集））。对于所有的平台，我们还应当启用快速浮点模型/fp:fast（**Project Properties**（工程属性）→**Configuration Properties**（配置属性）→**C/C++**→**Code Generation**（代码生成）→**Floating Point Model**（浮点模型））。而对于x64平台来说，我们却不必开启SSE2指令集，这是因为所有的x64 CPU对此均有支持。

1.6.1 向量类型

在DirectXMath库中，核心的向量类型是`XMVECTOR`，它将被映射到SIMD硬件寄存器。通过SIMD指令的配合，利用这种具有128位的类型能一次性处理4个32位的浮点数。在开启SSE2后，此类型在x86和x64平台的定义是：

```
typedef __m128 XMVECTOR;
```

这里的__m128是一种特殊的SIMD类型（定义见xmmintrin.h）。在计算向量的过程中，必须通过此类型才可充分地利用SIMD技术。正如前文所述，我们将通过SIMD技术来处理2D和3D向量运算，而计算过程中用不到的向量分量则将它置零并忽略。

XMVECTOR类型的数据需要按16字节对齐，这对于局部变量和全局变量而言都是自动实现的。至于类中的数据成员，建议分别使用XMFLOAT2（2D向量）、XMFLOAT3（3D向量）和XMFLOAT4（4D向量）类型来加以代替。这些结构体的定义如下[7]：

```
struct XMFLOAT2
{
  float x;
  float y;

  XMFLOAT2() {}
  XMFLOAT2(float _x, float _y) : x(_x), y(_y) {}
  explicit XMFLOAT2(_In_reads_(2) const float *pArray) :
    x(pArray[0]), y(pArray[1]) {}

  XMFLOAT2& operator= (const XMFLOAT2& Float2)
  { x = Float2.x; y = Float2.y; return *this; }
};


struct XMFLOAT3
{
  float x;
  float y;
  float z;

  XMFLOAT3() {}
  XMFLOAT3(float _x, float _y, float _z) : x(_x), y(_y), z(_z) {}
  explicit XMFLOAT3(_In_reads_(3) const float *pArray) :
    x(pArray[0]), y(pArray[1]), z(pArray[2]) {}

  XMFLOAT3& operator= (const XMFLOAT3& Float3)
  { x = Float3.x; y = Float3.y; z = Float3.z; return *this; }
};
```

```
struct XMFLOAT4
{
  float x;
  float y;
  float z;
  float w;

  XMFLOAT4() {}
  XMFLOAT4(float _x, float _y, float _z, float _w) :
    x(_x), y(_y), z(_z), w(_w) {}
  explicit XMFLOAT4(_In_reads_(4) const float *pArray) :
    x(pArray[0]), y(pArray[1]), z(pArray[2]), w(pArray[3]) {}

  XMFLOAT4& operator= (const XMFLOAT4& Float4)
  { x = Float4.x; y = Float4.y; z = Float4.z; w = Float4.w; return
    *this; }
};
```

但是，如果直接把上述这些类型用于计算，却依然不能充分发挥出SIMD技术的高效特性。为此，我们还需要将这些类型的实例转换为`XMVECTOR`类型。转换的过程可以通过DirectXMath库的加载函数（loading function）实现。相反地，DirectXMath库也提供了用来将`XMVECTOR`类型转换为`XMFLOAT`*n*类型的存储函数（storage function）。

总结一下：

1．局部变量或全局变量用`XMVECTOR`类型。

2．对于类中的数据成员，使用`XMFLOAT2`、`XMFLOAT3`和`XMFLOAT4`类型。

3．在运算之前，通过加载函数将`XMFLOAT`*n*类型转换为`XMVECTOR`

类型。

4. 用XMVECTOR实例来进行运算。

5. 通过存储函数将XMVECTOR类型转换为XMFLOAT*n*类型。

1.6.2 加载方法和存储方法

用下面的方法将数据从XMFLOAT*n*类型加载到XMVECTOR类型：

```
// 将数据从XMFLOAT2类型中加载到XMVECTOR类型
XMVECTOR XM_CALLCONV XMLoadFloat2(const XMFLOAT2 *pSource);

// 将数据从XMFLOAT3类型中加载到XMVECTOR类型
XMVECTOR XM_CALLCONV XMLoadFloat3(const XMFLOAT3 *pSource);

// 将数据从XMFLOAT4类型中加载到XMVECTOR类型
XMVECTOR XM_CALLCONV XMLoadFloat4(const XMFLOAT4 *pSource);
```

用下面的方法可将数据从XMVECTOR类型存储到XMFLOAT*n*类型：

```
// 将数据从XMVECTOR类型中存储到XMFLOAT2类型
void XM_CALLCONV XMStoreFloat2(XMFLOAT2 *pDestination, FXMVECTOR V);

// 将数据从XMVECTOR类型中存储到XMFLOAT3类型
void XM_CALLCONV XMStoreFloat3(XMFLOAT3 *pDestination, FXMVECTOR V);

// 将数据从XMVECTOR类型中存储到XMFLOAT4类型
void XM_CALLCONV XMStoreFloat4(XMFLOAT4 *pDestination, FXMVECTOR V);
```

当我们只希望从XMVECTOR实例中得到某一个向量分量或将某一向量分量转换为XMVECTOR类型时，相关的存取方法如下：

```
float XM_CALLCONV XMVectorGetX(FXMVECTOR V);
float XM_CALLCONV XMVectorGetY(FXMVECTOR V);
```

```
float XM_CALLCONV XMVectorGetZ(FXMVECTOR V);
float XM_CALLCONV XMVectorGetW(FXMVECTOR V);

XMVECTOR XM_CALLCONV XMVectorSetX(FXMVECTOR V, float x);
XMVECTOR XM_CALLCONV XMVectorSetY(FXMVECTOR V, float y);
XMVECTOR XM_CALLCONV XMVectorSetZ(FXMVECTOR V, float z);
XMVECTOR XM_CALLCONV XMVectorSetW(FXMVECTOR V, float w);
```

1.6.3　参数的传递

为了提高效率，可以将XMVECTOR类型的值作为函数的参数，直接传送至SSE/SSE2寄存器（register）里，而不存于栈（stack）内。以此方式传递的参数数量取决于用户使用的平台（例如，32位的Windows系统、64位的Windows系统及Windows RT系统所能传递的参数数量都各不相同）和编译器。因此，为了使代码更具通用性，不受具体平台、编译器的影响，我们将利用FXMVECTOR、GXMVECTOR、HXMVECTOR和CXMVECTOR类型来传递XMVECTOR类型的参数。基于特定的平台和编译器，它们会被自动地定义为适当的类型。此外，一定要把调用约定注解XM_CALLCONV加在函数名之前，它会根据编译器的版本确定出对应的调用约定属性。

传递XMVECTOR参数的规则如下：

1．前3个XMVECTOR参数应当用类型FXMVECTOR；

2．第4个XMVECTOR参数应当用类型GXMVECTOR；

3．第5、6个XMVECTOR参数应当用类型HXMVECTOR；

4. 其余的XMVECTOR参数应当用类型CXMVECTOR。

下面详解这些类型在32位Windows平台和编译器（编译器需要支持__fastcall和新增的__vectorcall调用约定）上的定义：

```
// 在32位的Windows系统上，编译器将根据__fastcall调用约定将前3个
// XMVECTOR参数传递到寄存器中，而把其余参数都存在栈上
typedef const XMVECTOR FXMVECTOR;
typedef const XMVECTOR& GXMVECTOR;
typedef const XMVECTOR& HXMVECTOR;
typedef const XMVECTOR& CXMVECTOR;

// 在32位的Windows系统上，编译器将通过__vectorcall调用约定将前6个
// XMVECTOR参数传递到寄存器中，而把其余参数均存在栈上
typedef const XMVECTOR FXMVECTOR;
typedef const XMVECTOR GXMVECTOR;
typedef const XMVECTOR HXMVECTOR;
typedef const XMVECTOR& CXMVECTOR;
```

对于这些类型在其他平台的定义细节，可参见DirectXMath库文档中“Library Internals（库的内部细节）”下的“Calling Conventions（调用约定）”部分[DirectXMath]。构造函数（constructor）方法对于这些规则来讲却是个例外。[DirectXMath]建议，在编写构造函数时，前3个XMVECTOR参数用FXMVECTOR类型，其余XMVECTOR参数则用CXMVECTOR类型。另外，对于构造函数不要使用XM_CALLCONV注解。

以下示例截取自DirectXMath库的源代码：

```
inline XMMATRIX XM_CALLCONV XMMatrixTransformation(
  FXMVECTOR ScalingOrigin,
  FXMVECTOR ScalingOrientationQuaternion,
  FXMVECTOR Scaling,
  GXMVECTOR RotationOrigin,
  HXMVECTOR RotationQuaternion,
  HXMVECTOR Translation);
```

此函数有6个XMVECTOR参数，根据参数传递法则，前3个参数用FXMVECTOR类型，第4个参数用GXMVECTOR类型，第5个和第6个参数则用HXMVECTOR类型。

在XMVECTOR类型的参数之间，我们也可以掺杂其他非XMVECTOR类型的参数。此时，XMVECTOR参数的规则依然适用，而在统计XMVECTOR参数的数量时，会对其他类型的参数视若无睹。例如，在下列函数中，前3个XMVECTOR参数的类型依旧为FXMVECTOR，第4个XMVECTOR参数的类型仍为GXMVECTOR。

```
inline XMMATRIX XM_CALLCONV XMMatrixTransformation2D(
  FXMVECTOR ScalingOrigin,
  float    ScalingOrientation,
  FXMVECTOR Scaling,
  FXMVECTOR RotationOrigin,
  float    Rotation,
  GXMVECTOR Translation);
```

传递XMVECTOR参数的规则仅适用于“输入”参数。“输出”的XMVECTOR参数（即XMVECTOR&或XMVECTOR*）则不会占用SSE/SSE2寄存器，所以它们的处理方式与非XMVECTOR类型的参数一致。

1.6.4 常向量

XMVECTOR类型的常量实例应当用XMVECTORF32类型来表示。在DirectX SDK中的CascadedShadowMaps11示例内就可见到这种类型的应用：

```
static const XMVECTORF32 g_vHalfVector = { 0.5f, 0.5f, 0.5f, 0.5f };
```

```
static const XMVECTORF32 g_vZero = { 0.0f, 0.0f, 0.0f, 0.0f };

XMVECTORF32 vRightTop = {
vViewFrust.RightSlope,
vViewFrust.TopSlope,
1.0f,1.0f
};

XMVECTORF32 vLeftBottom = {
vViewFrust.LeftSlope,
vViewFrust.BottomSlope,
1.0f,1.0f
};
```

基本上，在我们运用初始化语法的时候就要使用XMVECTORF32类型。

XMVECTORF32是一种按16字节对齐的结构体，数学库中还提供了将它转换至XMVECTOR类型的运算符。其定义如下：

```
// 将常向量转换为其他类型的运算符
__declspec(align(16)) struct XMVECTORF32
{
  union
  {
    float f[4];
    XMVECTOR v;
  };

  inline operator XMVECTOR() const { return v; }
  inline operator const float*() const { return f; }
#if !defined(_XM_NO_INTRINSICS_) && defined(_XM_SSE_INTRINSICS_)
  inline operator __m128i() const { return _mm_castps_si128(v); }
  inline operator __m128d() const { return _mm_castps_pd(v); }
#endif
};
```

另外，也可以通过XMVECTORU32类型来创建由整型数据构成的XMVECTOR常向量：

```
static const XMVECTORU32 vGrabY = {
0x00000000,0xFFFFFFFF,0x00000000,0x00000000
};
```

1.6.5 重载运算符

XMVECTOR类型针对向量的加法运算、减法运算和标量乘法运算，都分别提供了对应的重载运算符。

```
XMVECTOR  XM_CALLCONV   operator+ (FXMVECTOR V);
XMVECTOR  XM_CALLCONV   operator- (FXMVECTOR V);

XMVECTOR&  XM_CALLCONV   operator+==(XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR&  XM_CALLCONV   operator-= (XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR&  XM_CALLCONV   operator*= (XMVECTOR& V1, FXMVECTOR V2);
XMVECTOR&  XM_CALLCONV   operator/= (XMVECTOR& V1, FXMVECTOR V2);

XMVECTOR&  operator*= (XMVECTOR& V, float S);
XMVECTOR&  operator/= (XMVECTOR& V, float S);

XMVECTOR  XM_CALLCONV   operator+ (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR  XM_CALLCONV   operator- (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR  XM_CALLCONV   operator* (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR  XM_CALLCONV   operator/ (FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR  XM_CALLCONV   operator* (FXMVECTOR V, float S);
XMVECTOR  XM_CALLCONV   operator* (float S, FXMVECTOR V);
XMVECTOR  XM_CALLCONV   operator/ (FXMVECTOR V, float S);
```

1.6.6 杂项

DirectXMath库定义了一组与π有关的常用数学常量近似值：

```
const float XM_PI     =   3.141592654f;
const float XM_2PI    =   6.283185307f;
const float XM_1DIVPI     =  0.318309886f;
const float XM_1DIV2PI    =  0.159154943f;
const float XM_PIDIV2     =  1.570796327f;
```

```
const float XM_PIDIV4     =  0.785398163f;
```

另外，它用下列内联函数实现了弧度和角度间的互相转化：

```
inline float XMConvertToRadians(float fDegrees)
{ return fDegrees * (XM_PI / 180.0f); }
inline float XMConvertToDegrees(float fRadians)
{ return fRadians * (180.0f / XM_PI); }
```

DirectXMath库还定义了求出两个数间较大值及较小值的函数：

```
template<class T> inline T XMMin(T a, T b) { return (a < b) ? a : b; }
template<class T> inline T XMMax(T a, T b) { return (a > b) ? a : b; }
```

1.6.7　Setter函数

DirectXMath库提供了下列函数，以设置XMVECTOR类型中的数据：

```
// 返回零向量0
XMVECTOR XM_CALLCONV XMVectorZero();

// 返回向量(1, 1, 1, 1)
XMVECTOR XM_CALLCONV XMVectorSplatOne();

// 返回向量(x, y, z, w)
XMVECTOR XM_CALLCONV XMVectorSet(float x, float y, float z, float w);

// 返回向量(Value, Value, Value, Value)
XMVECTOR XM_CALLCONV XMVectorReplicate(float Value);

// 返回向量(v_x, v_x, v_x, v_x)
XMVECTOR XM_CALLCONV XMVectorSplatX(FXMVECTOR V);

// 返回向量(v_y, v_y, v_y, v_y)
XMVECTOR XM_CALLCONV XMVectorSplatY(FXMVECTOR V);

// 返回向量(v_z, v_z, v_z, v_z)
XMVECTOR XM_CALLCONV XMVectorSplatZ(FXMVECTOR V);
```

下列的示例程序详细地解释了上面大多数函数的用法：

```
#include <windows.h> // 为了使XMVerifyCPUSupport函数返回正确值
#include <DirectXMath.h>
#include <DirectXPackedVector.h>
#include <iostream>
using namespace std;
using namespace DirectX;
using namespace DirectX::PackedVector;

// 重载"<<"运算符，这样就可以通过cout函数输出XMVECTOR对象
ostream& XM_CALLCONV operator<<(ostream& os, FXMVECTOR v)
{
  XMFLOAT3 dest;
  XMStoreFloat3(&dest, v);

  os << "(" << dest.x << ", " << dest.y << ", " << dest.z << ")";
  return os;
}

int main()
{
  cout.setf(ios_base::boolalpha);

  // 检查是否支持SSE2指令集 (Pentium4, AMD K8及其后续版本的处理器)
  if (!XMVerifyCPUSupport())
  {
    cout << "directx math not supported" << endl;
    return 0;
  }

  XMVECTOR p = XMVectorZero();
  XMVECTOR q = XMVectorSplatOne();
  XMVECTOR u = XMVectorSet(1.0f, 2.0f, 3.0f, 0.0f);
  XMVECTOR v = XMVectorReplicate(-2.0f);
  XMVECTOR w = XMVectorSplatZ(u);

  cout << "p = " << p << endl;
  cout << "q = " << q << endl;
  cout << "u = " << u << endl;
  cout << "v = " << v << endl;
  cout << "w = " << w << endl;

  return 0;
}
```

上述示例程序的输出结果如图1.18所示。

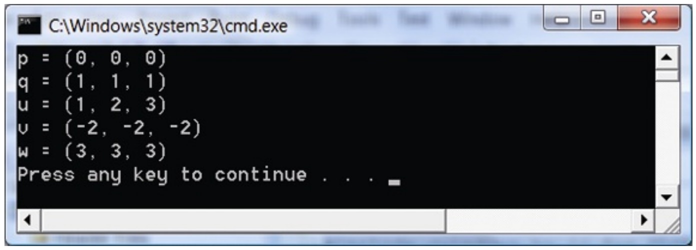


图1.18　示例程序输出的结果

1.6.8　向量函数

DirectXMath库提供了下面的函数来执行各种向量运算。我们主要围绕3D向量的运算函数进行讲解，类似的运算还有2D和4D版本。除了表示维度的数字不同以外，这几种版本的函数名皆同。

```
XMVECTOR XM_CALLCONV XMVector3Length(       // 返回||v||
  FXMVECTOR V);                             // 输入向量v

XMVECTOR XM_CALLCONV XMVector3LengthSq(     //返回||v||^2
  FXMVECTOR V);                             // 输入向量v

XMVECTOR XM_CALLCONV XMVector3Dot(          // 返回v1·v2
  FXMVECTOR V1,                             // 输入向量v1
  FXMVECTOR V2);                            // 输入向量v2

XMVECTOR XM_CALLCONV XMVector3Cross(        // 返回v1×v2
  FXMVECTOR V1,                             // 输入向量v1
  FXMVECTOR V2);                            // 输入向量v2

XMVECTOR XM_CALLCONV XMVector3Normalize(    // 返回v/||v||
  FXMVECTOR V);                             // 输入向量v

XMVECTOR XM_CALLCONV XMVector3Orthogonal(   // 返回一个正交于v的向量
  FXMVECTOR V);                             // 输入向量v

XMVECTOR XM_CALLCONV
```

```
XMVector3AngleBetweenVectors(              // 返回v1和v2之间的夹角
  FXMVECTOR V1,                            // 输入向量v1
  FXMVECTOR V2);                           // 输入向量v2

void XM_CALLCONV XMVector3ComponentsFromNormal(
  XMVECTOR* pParallel,                     // 返回projn(v)
  XMVECTOR* pPerpendicular,                // 返回perpn(v)
  FXMVECTOR V,                             // 输入向量v
  FXMVECTOR Normal);                       // 输入规范化向量n

bool XM_CALLCONV XMVector3Equal(      // 返回v1 == v2?
  FXMVECTOR V1,                       // 输入向量v1
  FXMVECTOR V2);                      // 输入向量v2

bool XM_CALLCONV XMVector3NotEqual(   // 返回v1≠v2
  FXMVECTOR V1,                       // 输入向量v1
  FXMVECTOR V2);                      // 输入向量v2
```

注意 Note

可以看到，即使在数学上计算的结果是标量（如点积$k = \mathbf{v}_1 \cdot \mathbf{v}_2$），但这些函数所返回的类型依旧是XMVECTOR，而得到的标量结果则被复制到XMVECTOR中的各个分量之中。例如点积，此函数返回的向量为（$\mathbf{v}_1 \cdot \mathbf{v}_2, \mathbf{v}_1 \cdot \mathbf{v}_2, \mathbf{v}_1 \cdot \mathbf{v}_2, \mathbf{v}_1 \cdot \mathbf{v}_2$）。这样做的原因之一是：将标量和SIMD向量的混合运算次数降到最低，使用户除了自定义的计算之外全程都使用SIMD技术，以提升计算效率。

下面的程序演示了如何使用上述大部分函数，其中还示范了一些重载运算符的用法：

```
#include <windows.h> // 为了使XMVerifyCPUSupport函数返回正确值
#include <DirectXMath.h>
#include <DirectXPackedVector.h>
#include <iostream>
using namespace std;
using namespace DirectX;
using namespace DirectX::PackedVector;

// 对"<<"运算符进行重载，这样就可以通过cout函数输出XMVECTOR对象
ostream& XM_CALLCONV operator<<(ostream& os, FXMVECTOR v)
{
  XMFLOAT3 dest;
  XMStoreFloat3(&dest, v);

  os << "(" << dest.x << ", " << dest.y << ", " << dest.z << ")";
  return os;
}

int main()
{
  cout.setf(ios_base::boolalpha);

  // 检查是否支持SSE2指令集 (Pentium4, AMD K8及其后续版本的处理器)
  if (!XMVerifyCPUSupport())
  {
    cout << "directx math not supported" << endl;
    return 0;
  }

  XMVECTOR n = XMVectorSet(1.0f, 0.0f, 0.0f, 0.0f);
  XMVECTOR u = XMVectorSet(1.0f, 2.0f, 3.0f, 0.0f);
  XMVECTOR v = XMVectorSet(-2.0f, 1.0f, -3.0f, 0.0f);
  XMVECTOR w = XMVectorSet(0.707f, 0.707f, 0.0f, 0.0f);

  // 向量加法：利用XMVECTOR类型的加法运算符+ +
  XMVECTOR a = u + v;

  // 向量减法：利用XMVECTOR类型的减法运算符-
  XMVECTOR b = u - v;

  // 标量乘法：利用XMVECTOR类型的标量乘法运算符*
  XMVECTOR c = 10.0f*u;

  // ||u||
  XMVECTOR L = XMVector3Length(u);

  // d = u / ||u||
```

```
    XMVECTOR d = XMVector3Normalize(u);

    // s = u dot v
    XMVECTOR s = XMVector3Dot(u, v);

    // e = u x v
    XMVECTOR e = XMVector3Cross(u, v);

    // 求出proj_n(w)和perp_n(w)
    XMVECTOR projW;
    XMVECTOR perpW;
    XMVector3ComponentsFromNormal(&projW, &perpW, w, n);

    // projW + perpW == w?
    bool equal = XMVector3Equal(projW + perpW, w) != 0;
    bool notEqual = XMVector3NotEqual(projW + perpW, w) != 0;

    // projW与perpW之间的夹角应为90度
    XMVECTOR angleVec = XMVector3AngleBetweenVectors(projW, perpW);
    float angleRadians = XMVectorGetX(angleVec);
    float angleDegrees = XMConvertToDegrees(angleRadians);

    cout << "u            = " << u << endl;
    cout << "v            = " << v << endl;
    cout << "w            = " << w << endl;
    cout << "n            = " << n << endl;
    cout << "a = u + v    = " << a << endl;
    cout << "b = u - v    = " << b << endl;
    cout << "c = 10 * u   = " << c << endl;
    cout << "d = u / ||u|| = " << d << endl;
    cout << "e = u x v    = " << e << endl;
    cout << "L = ||u||    = " << L << endl;
    cout << "s = u.v      = " << s << endl;
    cout << "projW        = " << projW << endl;
    cout << "perpW        = " << perpW << endl;
    cout << "projW + perpW == w = " << equal << endl;
    cout << "projW + perpW != w = " << notEqual << endl;
    cout << "angle        = " << angleDegrees << endl;

    return 0;
}
```

上述示例程序的输出结果如图1.19所示。

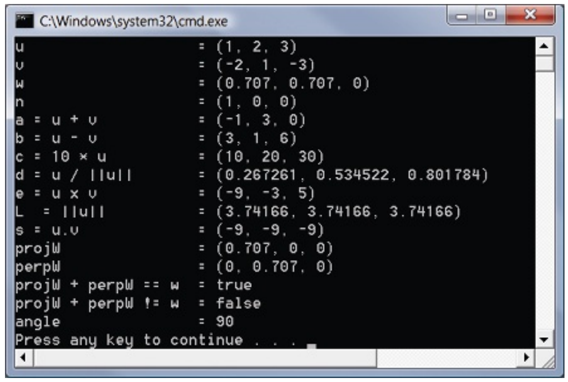


图1.19　示例程序的输出结果

注意 Note

DirectXMath库也提供了一些估算方法，精度低但速度快。如果愿意为了速度而牺牲一些精度，则可以使用它们。下面是两个估算方法的例子。

```
XMVECTOR XM_CALLCONV XMVector3LengthEst(      // 返回估算值||v||
  FXMVECTOR V);                                // 输入v

XMVECTOR XM_CALLCONV XMVector3NormalizeEst(   // 返回估算值v/||v||
  FXMVECTOR V);                               // 输入v
```

1.6.9　浮点数误差

在用计算机处理与向量有关的工作时，我们应当了解以下的内容。在比较浮点数时，一定要注意浮点数存在的误差。我们认为相等的两个

浮点数可能会因此而有细微的差别。例如，已知在数学上规范化向量的长度为1，但是在计算机程序中的表达上，向量的长度只能接近于1。此外，在数学中，对于任意实数p有$1^p = 1$。但是，当只能在数值上逼近1时，随着幂p的增加，所求近似值的误差也在逐渐增大。由此可见，数值误差是可积累的。下面这个小程序可印证这些观点：

```
#include <windows.h> // 为了使XMVerifyCPUSupport函数返回正确值
#include <DirectXMath.h>
#include <DirectXPackedVector.h>
#include <iostream>
using namespace std;
using namespace DirectX;
using namespace DirectX::PackedVector;

int main()
{
  cout.precision(8);

  // 检查是否支持SSE2指令集 (Pentium4, AMD K8及其后续版本的处理器)
  if (!XMVerifyCPUSupport())
  {
    cout << "directx math not supported" << endl;
    return 0;
  }

  XMVECTOR u = XMVectorSet(1.0f, 1.0f, 1.0f, 0.0f);
  XMVECTOR n = XMVector3Normalize(u);

  float LU = XMVectorGetX(XMVector3Length(n));

  // 在数学上，此向量的长度应当为1。在计算机中的数值表达上也是如此吗？
  cout << LU << endl;
  if (LU == 1.0f)
    cout << "Length 1" << endl;
  else
    cout << "Length not 1" << endl;

  // 1的任意次方都是1。但是在计算机中，事实确实如此吗？
  float powLU = powf(LU, 1.0e6f);
  cout << "LU^(10^6) = " << powLU << endl;
}
```

上述示例程序的输出结果如图1.20所示。

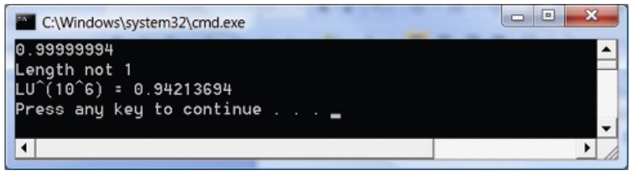


图1.20　示例程序输出的结果

为了弥补浮点数精确性上的不足，我们通过比较两个浮点数是否近似相等来加以解决。在比较的时候，我们需要定义一个Epsilon常量，它是个非常小的值，可为误差留下一定的“缓冲”余地。如果两个数相差的值小于Epsilon，我们就说这两个数是近似相等的。换句话说，Epsilon是针对浮点数的误差问题所指定的容差（tolerance）。下面的函数解释了如何利用Epsilon来检测两个浮点数是否相等：

```
const float Epsilon = 0.001f;
bool Equals(float lhs, float rhs)
{
    // lhs和rhs相差的值是否小于EPSILON?
    return fabs(lhs - rhs) < Epsilon ? true : false;
}
```

对此，DirectXMath库提供了XMVector3NearEqual函数，用于以Epsilon作为容差，测试比较的向量是否相等：

```
// 返回
//  abs(U.x – V.x) <= Epsilon.x &&
//  abs(U.y – V.y) <= Epsilon.y &&
//  abs(U.z – V.z) <= Epsilon.z
XMFINLINE bool XM_CALLCONV XMVector3NearEqual(
  FXMVECTOR U,
  FXMVECTOR V,
  FXMVECTOR Epsilon);
```

1.7 小结

1．向量可以用来模拟同时具有大小和方向的物理量。在几何学上，我们用有向线段表示向量。当向量平移至尾部与所在坐标系原点恰好重合的位置时，向量位于标准位置。一旦向量处于标准位置，我们便可以用向量头部相对于坐标系的坐标来作为它的数学描述。

2．假设有向量$\boldsymbol{u}=(u_x,u_y,u_z)$和向量$\boldsymbol{v}=(v_x,v_y,v_z)$，那么就能对它们进行下列向量计算。

（a）加法运算：$\boldsymbol{u}+\boldsymbol{v}=(u_x+v_x,u_y+v_y,u_z+v_z)$

（b）减法运算：$\boldsymbol{u}-\boldsymbol{v}=(u_x-v_x,u_y-v_y,u_z-v_z)$

（c）标量乘法运算：$k\boldsymbol{u}=(ku_x,ku_y,ku_z)$

（d）向量长度：$\|\boldsymbol{u}\|=\sqrt{x^2+y^2+z^2}$

（e）规范化：$\hat{\boldsymbol{u}}=\frac{\boldsymbol{u}}{\|\boldsymbol{u}\|}=\left(\frac{x}{\|\boldsymbol{u}\|},\frac{y}{\|\boldsymbol{u}\|},\frac{z}{\|\boldsymbol{u}\|}\right)$

（f）点积：$\boldsymbol{u}\cdot\boldsymbol{v}=\|\boldsymbol{u}\|\|\boldsymbol{v}\|\cos\boldsymbol{\theta}=u_xv_x+u_yv_y+u_zv_z$

（g）叉积：$\boldsymbol{u}\times\boldsymbol{v}=(u_yv_z-u_zv_y,u_zv_x-u_xv_z,u_xv_y-u_yv_x)$

3．用DirectXMath库的`XMVECTOR`类型来描述向量，这样就可以在代码中利用SIMD技术进行高效的运算。对于类中的数据成员来说，要

使用XMFLOAT2、XMFLOAT3和XMFLOAT4这些类表示向量，并通过加载和存储方法令数据在XMVECTOR类型与XMFLOAT*n*类型之间互相转化。另外，在使用常向量的初始化语法时，应当采用XMVECTORF32类型。

4．为了提高效率，当XMVECTOR类型的值被当作参数传入函数时，可以直接存入SSE/SSE2寄存器中而不是栈上。要令代码与平台无关，我们将使用FXMVECTOR、GXMVECTOR、HXMVECTOR和CXMVECTOR类型来传递XMVECTOR参数。传递XMVECTOR参数的规则为：前3个XMVECTOR参数应当用FXMVECTOR类型，第4个XMVECTOR参数用GXMVECTOR类型，第5个和第6个XMVECTOR参数用HXMVECTOR类型，而其余的XMVECTOR类型参数则用CXMVECTOR类型。

5．XMVECTOR类重载了一些运算符用来实现向量的加法、减法和标量乘法。另外，DirectXMath库还提供了下面一些实用的函数，用于计算向量的模、模的平方、两个向量的点积、两个向量的叉积以及对向量进行规范化处理：

```
XMVECTOR XM_CALLCONV XMVector3Length(FXMVECTOR V);
XMVECTOR XM_CALLCONV XMVector3LengthSq(FXMVECTOR V);
XMVECTOR XM_CALLCONV XMVector3Dot(FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR XM_CALLCONV XMVector3Cross(FXMVECTOR V1, FXMVECTOR V2);
XMVECTOR XM_CALLCONV XMVector3Normalize(FXMVECTOR V);
```

1.8　练习

1．设向量$\boldsymbol{u}=(1,2)$和向量$\boldsymbol{v}=(3,-4)$。写出下列各式的演算过程，并在2D坐标系内画出相应的向量。

（a）$\boldsymbol{u}+\boldsymbol{v}$

（b）$\boldsymbol{u}-\boldsymbol{v}$

（c）$2\boldsymbol{u}+\frac{1}{2}\boldsymbol{v}$

（d）$-2\boldsymbol{u}+\boldsymbol{v}$

2．设向量$\boldsymbol{u}=(-1,3,2)$和向量$\boldsymbol{v}=(3,-4,1)$。写出下列问题的解答过程。

（a）$\boldsymbol{u}+\boldsymbol{v}$

（b）$\boldsymbol{u}-\boldsymbol{v}$

（c）$3\boldsymbol{u}+2\boldsymbol{v}$

（d）$-2\boldsymbol{u}+\boldsymbol{v}$

3．本习题展示了向量代数与实数所共有的一些计算性质（注意，以下清单中所列举的性质并不完整）。假设有向量$\boldsymbol{u}=(u_x,u_y,u_z)$、$\boldsymbol{v}=(v_x,v_y,v_z)$和$\boldsymbol{w}=(w_x,w_y,w_z)$，另有标量$c$和$k$，请证明下列向量性质。

（a）$\boldsymbol{u}+\boldsymbol{v}=\boldsymbol{v}+\boldsymbol{u}$（加法交换律）

（b）$\boldsymbol{u}+(\boldsymbol{v}+\boldsymbol{w})=(\boldsymbol{u}+\boldsymbol{v})+\boldsymbol{w}$（加法结合律）

（c）$(ck)\boldsymbol{u}=c(k\boldsymbol{u})$（标量乘法的结合律）

（d）$k(\boldsymbol{u}+\boldsymbol{v})=k\boldsymbol{u}+k\boldsymbol{v}$（分配律1）

（e）$\boldsymbol{u}(k+c)=k\boldsymbol{u}+c\boldsymbol{u}$（分配律2）

提示

仅利用向量运算的定义和实数的性质即可完成证明。例如，

$$\begin{aligned}(ck)\boldsymbol{u}&=(ck)\left(u_x,u_y,u_z\right)\\&=((ck)u_x,(ck)u_y,(ck)u_z)\\&=\left(c\left(ku_x\right),c\left(ku_y\right),c\left(ku_z\right)\right)\\&=c\left(ku_x,ku_y,ku_z\right)\\&=c(k\boldsymbol{u})\end{aligned}$$

4．根据等式$2[(1,2,3)-\boldsymbol{x}]-(-2,0,4)=-2(1,2,3)$，求其中的向量$\boldsymbol{x}$。

5．设向量$\boldsymbol{u}=(-1,3,2)$和向量$\boldsymbol{v}=(3,-4,1)$。对$\boldsymbol{u}$和$\boldsymbol{v}$进行规范化处理。

6．设k为标量，向量$\boldsymbol{u}=(u_x,u_y,u_z)$。求证$||k\boldsymbol{u}||=||k||||\boldsymbol{u}||$。

7．下列各组向量中，$\boldsymbol{u}$与$\boldsymbol{v}$之间的夹角是直角、锐角还是钝角？

（a）$\boldsymbol{u}=(1,1,1)$，$\boldsymbol{v}=(2,3,4)$

（b）$\boldsymbol{u}=(1,1,0)$，$\boldsymbol{v}=(-2,2,0)$

（c）$\boldsymbol{u}=(-1,-1,-1)$，$\boldsymbol{v}=(3,1,0)$

8．设向量$\boldsymbol{u}=(-1,3,2)$和向量$\boldsymbol{v}=(3,-4,1)$。计算$\boldsymbol{u}$和$\boldsymbol{v}$之间的夹角θ

。

9．设向量$\boldsymbol{u}=(u_x,u_y,u_z)$、$\boldsymbol{v}=(v_x,v_y,v_z)$和$\boldsymbol{w}=(w_x,w_y,w_z)$，且$c$和$k$为标量。证明下列点积性质。

（a）$\boldsymbol{u}\cdot\boldsymbol{v}=\boldsymbol{v}\cdot\boldsymbol{u}$

（b）$\boldsymbol{u}\cdot(\boldsymbol{v}+\boldsymbol{w})=\boldsymbol{u}\cdot\boldsymbol{v}+\boldsymbol{u}\cdot\boldsymbol{w}$

（c）$k(\boldsymbol{u}\cdot\boldsymbol{v})=(k\boldsymbol{u})\cdot\boldsymbol{v}=\boldsymbol{u}\cdot(k\boldsymbol{v})$

（d）$\boldsymbol{v}\cdot\boldsymbol{v}=||\boldsymbol{v}||^2$

（e）$\boldsymbol{0}\cdot\boldsymbol{v}=0$

提示

仅利用前文介绍的各种定义即可证明，例如，

$$\begin{aligned}\boldsymbol{v}\cdot\boldsymbol{v} &= v_x v_x + v_y v_y + v_z v_z \\ &= v_x^2 + v_y^2 + v_z^2 \\ &= (\sqrt{v_x^2 + v_y^2 + v_z^2})^2 \\ &= ||\boldsymbol{v}||^2\end{aligned}$$

10．利用余弦定理（$c^2 = a^2 + b^2 - 2ab\cos\theta$，其中$a$、$b$、$c$分别是三角形3条边的边长，$\theta$为$a$与$b$之间的夹角）来证明：

$$u_x v_x + u_y v_y + u_z v_z = ||\boldsymbol{u}||\,||\boldsymbol{v}||\cos\theta$$

提示

参考图1.9，设$c^2 = ||\boldsymbol{u} - \boldsymbol{v}||^2$，以及$b^2 = ||\boldsymbol{v}||^2$，再运用上一个习题中得到的点积性质即可。

11．设向量$\boldsymbol{n} = (-2, 1)$。将向量$\boldsymbol{g} = (0, -9.8)$分解为两个相互正交的向量之和，使它们一个平行于$\boldsymbol{n}$、一个正交于$\boldsymbol{n}$。最后，在同一2D坐标系中画出这些向量。

12．设向量$\boldsymbol{u} = (-2, 1, 4)$和向量$\boldsymbol{v} = (3, -4, 1)$。求向量$\boldsymbol{w} = \boldsymbol{u}\times\boldsymbol{v}$，再证明$\boldsymbol{w}\cdot\boldsymbol{u} = 0$及$\boldsymbol{w}\cdot\boldsymbol{v} = 0$。

13．设$\boldsymbol{A}=(0,0,0)$，$\boldsymbol{B}=(0,1,3)$和$\boldsymbol{C}=(5,1,0)$三点在某坐标系中定义了一个三角形。求出一正交于此三角形的向量。

提示

先求出位于三角形任意两条边上的两个向量，再对它们进行叉积运算即可。

14．证明$\|\boldsymbol{u}\times\boldsymbol{v}\|=\|\boldsymbol{u}\|\|\boldsymbol{v}\|\sin\theta$。

提示

从$\|\boldsymbol{u}\|\|\boldsymbol{v}\|\sin\theta$一侧开始证明，先利用三角恒等式$\cos^2\theta+\sin^2\theta=1\Rightarrow\sin\theta=\sqrt{1-\cos^2\theta}$，再运用式（1.4）。

15．证明：由向量$\boldsymbol{u}$和向量$\boldsymbol{v}$张成的平行四边形面积为$\|\boldsymbol{u}\times\boldsymbol{v}\|$，如图1.21所示[8]。

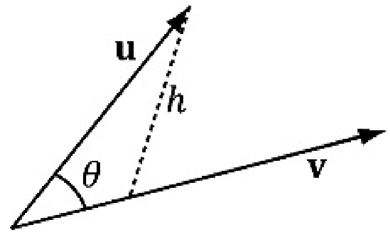


图1.21　由向量$\boldsymbol{u}$和向量$\boldsymbol{v}$张成的平行四边形。此平行四边形的底为$||\boldsymbol{v}||$且高为h

16．举例证明：存在3D向量$\boldsymbol{u}$、$\boldsymbol{v}$和$\boldsymbol{w}$，满足$\boldsymbol{u} \times (\boldsymbol{v} \times \boldsymbol{w}) \neq (\boldsymbol{u} \times \boldsymbol{v}) \times \boldsymbol{w}$。这说明叉积一般不满足结合律。

提示

考虑这个简单的向量组合：$\boldsymbol{i} = (1,0,0)$，$\boldsymbol{j} = (0,1,0)$和$\boldsymbol{k} = (0,0,1)$。

17．证明两个非零且相互平行向量的叉积为零向量，即$\boldsymbol{u} \times k\boldsymbol{u} = \mathbf{0}$。

提示

直接利用叉积定义即可。

18．利用格拉姆—施密特正交化方法，令向量集 $\{(1,0,0),(1,5,0),(2,1,-4)\}$规范正交化。

19．思考下面的程序及其输出结果（见图1.22）。猜测其中每个XMVector*函数的功能。然后在DirectXMath文档中，查阅每个函数的相关信息[9]。

```
#include <windows.h> // 为了使用XMVerifyCPUSupport函数返回正确值
#include <DirectXMath.h>
#include <DirectXPackedVector.h>
#include <iostream>
using namespace std;
using namespace DirectX;
using namespace DirectX::PackedVector;

// 重载"<<"运算符，这样便可以使用cout输出XMVECTOR对象
ostream& XM_CALLCONV operator<<(ostream& os, FXMVECTOR v)
{
  XMFLOAT4 dest;
  XMStoreFloat4(&dest, v);

  os << "(" << dest.x << ", " << dest.y << ", "
     << dest.z << ", " << dest.w << ")";
  return os;
}

int main()
{
  cout.setf(ios_base::boolalpha);

  // 检查是否支持SSE2指令集 (Pentium4, AMD K8及其后续版本的处理器)
  if (!XMVerifyCPUSupport())
  {
    cout << "directx math not supported" << endl;
    return 0;
  }

  XMVECTOR p = XMVectorSet(2.0f, 2.0f, 1.0f, 0.0f);
  XMVECTOR q = XMVectorSet(2.0f, -0.5f, 0.5f, 0.1f);
  XMVECTOR u = XMVectorSet(1.0f, 2.0f, 4.0f, 8.0f);
  XMVECTOR v = XMVectorSet(-2.0f, 1.0f, -3.0f, 2.5f);
  XMVECTOR w = XMVectorSet(0.0f, XM_PIDIV4, XM_PIDIV2, XM_PI);
```

```
    cout << "XMVectorAbs(v)         = " << XMVectorAbs(v) << endl;
    cout << "XMVectorCos(w)         = " << XMVectorCos(w) << endl;
    cout << "XMVectorLog(u)         = " << XMVectorLog(u) << endl;
    cout << "XMVectorExp(p)         = " << XMVectorExp(p) << endl;

    cout << "XMVectorPow(u, p)      = " << XMVectorPow(u, p) << endl;
    cout << "XMVectorSqrt(u)        = " << XMVectorSqrt(u) << endl;

    cout << "XMVectorSwizzle(u, 2, 2, 1, 3) = "
         << XMVectorSwizzle(u, 2, 2, 1, 3) << endl;
    cout << "XMVectorSwizzle(u, 2, 1, 0, 3) = "
         << XMVectorSwizzle(u, 2, 1, 0, 3) << endl;

    cout << "XMVectorMultiply(u, v)     = " << XMVectorMultiply(u, v) << end
l;
    cout << "XMVectorSaturate(q)        = " << XMVectorSaturate(q) << endl;
    cout << "XMVectorMin(p, v)          = " << XMVectorMin(p, v) << endl;
    cout << "XMVectorMax(p, v)          = " << XMVectorMax(p, v) << endl;

    return 0;
}
```

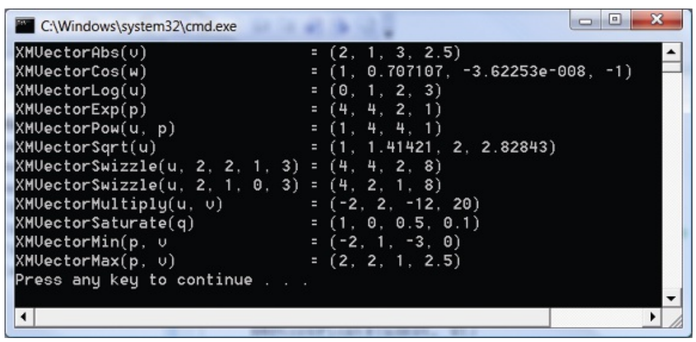


图1.22　上述程序输出的结果

---

[1]　“净”的对应英文为net，大抵表示为最终合成的总效果，如净方向即质点在不同力的作用下所移动的方向（也就是这几个作用力的合力方向）。后文同。

[2]　本书中所使用的术语“标架”（frame）、“参考系”（frame of

reference）、“空间”（space）和“坐标系”（coordinate system）皆表示相同的意义。——原作者注

[3]　准确地说还应考虑到气压因素。

[4]　这里所讲的都是推断坐标系各轴大致方向的办法，所谓“弯曲四指”意即找寻与“弯曲之前”垂直的坐标轴。下同。

[5]　毕达哥拉斯定理即勾股定理。西方文献常称勾股定理为毕达哥拉斯定理。

[6]　看到normalize这个词的各种译法就让我咬牙切齿！这个词在不同的学科里有着不同的译法，就算是同一学科的不同文献、不同词典的译法也是各异，如标准化、归一化、正常化、规格化、正态化、单位化……而且现在各种讨论中大多是统计学方面的，不同人给出的解释也各有差异，刨根问底也找不出图形学方面的译法。这就与“向量”相似，译作“矢量”也可，而且用这两种译法的书籍皆有。故现以数学文献和主流网站上的译法为准，基本上称区间、范围为“归一化”，名词向量或空间等译作“规范化”。当然，也有我水平不足之嫌。写这段话的目的其实就是希望读者不要过分拘泥于名词译法，个人以为只要在查看各种文献、与他人交流知道彼此在谈什么即可，其他地方也是如此。

[7]　DirectXMath.h头文件会随着DirectXMath库版本的更新而变更，因此可能会有若干细节与读者所用的版本不符。DirectXMath库中的其他文件也存在这种情况。此库当前的主要维护人是Chuck Walbourn。读者可以访问Chuck Walbourn的博客或GitHub网站以获得最新信息。

[8]　私以为平行四边形的高h应垂直于向量$\boldsymbol{v}$。

[9]　注意XMVectorCos函数的输出结果。

第2章　矩阵代数

在计算机3D图形学中，我们利用矩阵简洁地描述几何体的变换，例如缩放、旋转和平移。除此之外，还可借助矩阵将点或向量的坐标在不同的标架之间进行转换。在本章中，我们将探索与矩阵有关的数学知识。

学习目标：

1．理解矩阵及其相关运算的定义。

2．探究为何能将向量与矩阵的乘法视为一种线性组合。

3．学习单位矩阵、转置矩阵、行列式以及矩阵的逆等概念。

4．逐步熟悉DirectXMath库中提供的关于矩阵计算的类与函数的子集。

2.1　矩阵的定义

一个规模为$m \times n$的矩阵（matrix）$\boldsymbol{M}$，是由m行n列实数所构成的矩形阵列。[1]行数和列数的乘积表示了矩阵的维度。矩阵中的数字则称作元素（element）或元（entry）。通过双下标表示法$\boldsymbol{M}_{ij}$指定元素的行和列就可以确定出对应的矩阵元素，$\boldsymbol{M}_{ij}$表示的是矩阵中第i行、第j列的元素。

例2. 1

考察下列矩阵：

$$\boldsymbol{A} = \begin{bmatrix} 3.5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 2 & -5 & \sqrt{2} & 1 \end{bmatrix} \boldsymbol{B} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{bmatrix} \boldsymbol{u} = [u_1, u_2, u_3] \ \boldsymbol{v} = \begin{bmatrix} 1 \\ 2 \\ \sqrt{3} \\ \pi \end{bmatrix}$$

1．$\boldsymbol{A}$是一个4×4矩阵，$\boldsymbol{B}$是一个3×2矩阵，$\boldsymbol{u}$是一个1×3矩阵，$\boldsymbol{v}$是一个4×1矩阵。

2．我们通过双下标表示法$\boldsymbol{A}_{42}=-5$将矩阵$\boldsymbol{A}$中第4行第2列的元素指定为−5，并以$\boldsymbol{B}_{21}$表示矩阵$\boldsymbol{B}$中第2行第1列的这一元素。

3．$\boldsymbol{u}$与$\boldsymbol{v}$是两种特殊矩阵，分别只由一行元素或一列元素构成。由于它们常用于以矩阵的形式来表示一个向量（例如，我们可以自由地交替使用(x, y, z)与$[x, y, z]$这两种向量记法），因此有时候也分别称它们为行向量或列向量。观察可知，在表示行向量和列向量的矩阵元素时不必

再采用双下标——使用单下标记法即可。

在某些情况下，我们倾向于把矩阵的每一行都看作一个向量。例如，可以把矩阵写作：

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} \leftarrow \boldsymbol{A}_{1,*} \rightarrow \\ \leftarrow \boldsymbol{A}_{2,*} \rightarrow \\ \leftarrow \boldsymbol{A}_{3,*} \rightarrow \end{bmatrix}$$

其中，$\boldsymbol{A}_{1,*} = [A_{11}, A_{12}, A_{13}], \boldsymbol{A}_{2,*} = [A_{21}, A_{22}, A_{23}]$，$\boldsymbol{A}_{3,*} = [A_{31}, A_{32}, A_{33}]$。 在这种表达方式中，第一个索引表示特定的行，第二个索引“*”表示该行的整个行向量。而且，对于矩阵的列也有类似的定义：

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} \uparrow & \uparrow & \uparrow \\ \boldsymbol{A}_{*,1} & \boldsymbol{A}_{*,2} & \boldsymbol{A}_{*,3} \\ \downarrow & \downarrow & \downarrow \end{bmatrix}$$

其中

$$\boldsymbol{A}_{*,1} = \begin{bmatrix} A_{11} \\ A_{21} \\ A_{31} \end{bmatrix},\ \boldsymbol{A}_{*,2} = \begin{bmatrix} A_{12} \\ A_{22} \\ A_{32} \end{bmatrix},\ \boldsymbol{A}_{*,3} = \begin{bmatrix} A_{13} \\ A_{23} \\ A_{33} \end{bmatrix}$$

在这种表达方法中，第二个索引表示特定的列，第一个索引“*”表示该列的整个列向量。

现在来定义矩阵相等、加法运算、标量乘法运算和减法运算。

1．两个矩阵相等，当且仅当这两个矩阵的对应元素相等。为了加以比较，两者必有相同的行数和列数。

2．两个矩阵的加法运算，即将两者对应的元素相加。同理，只有行数和列数都分别相同的两个矩阵相加才有意义。

3．矩阵的标量乘法就是将一个标量依次与矩阵内的每个元素相乘。

4．利用矩阵的加法和标量乘法可以定义出矩阵的减法，即 $\boldsymbol{A}-\boldsymbol{B}=\boldsymbol{A}+(-1\cdot\boldsymbol{B})=\boldsymbol{A}+(-\boldsymbol{B})$。

例2. 2

设

$$\boldsymbol{A}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix},\ \boldsymbol{B}=\begin{bmatrix}6 & 2\\ 5 & -8\end{bmatrix},\ \boldsymbol{C}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix},\ \boldsymbol{D}=\begin{bmatrix}2 & 1 & -3\\ -6 & 3 & 0\end{bmatrix}$$

那么，

（i） $$\boldsymbol{A}+\boldsymbol{B}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix}+\begin{bmatrix}6 & 2\\ 5 & -8\end{bmatrix}=\begin{bmatrix}1+6 & 5+2\\ -2+5 & 3+(-8)\end{bmatrix}=\begin{bmatrix}7 & 7\\ 3 & -5\end{bmatrix}$$

（ii） $\boldsymbol{A}=\boldsymbol{C}$

（iii） $$3\boldsymbol{D}=3\begin{bmatrix}2 & 1 & -3\\ -6 & 3 & 0\end{bmatrix}=\begin{bmatrix}3(2) & 3(1) & 3(-3)\\ 3(-6) & 3(3) & 3(0)\end{bmatrix}=\begin{bmatrix}6 & 3 & -9\\ -18 & 9 & 0\end{bmatrix}$$

（iv） $$\boldsymbol{A}-\boldsymbol{B}=\begin{bmatrix}1 & 5\\ -2 & 3\end{bmatrix}-\begin{bmatrix}6 & 2\\ 5 & -8\end{bmatrix}=\begin{bmatrix}1-6 & 5-2\\ -2-5 & 3-(-8)\end{bmatrix}=\begin{bmatrix}-5 & 3\\ -7 & 11\end{bmatrix}$$

由于在矩阵的加法和标量乘法的运算过程中，是以元素为单位展开

计算的，所以它们实际上也分别从实数运算中继承了下列性质：

1. $\boldsymbol{A}+\boldsymbol{B}=\boldsymbol{B}+\boldsymbol{A}$ 加法交换律

2. $(\boldsymbol{A}+\boldsymbol{B})+\boldsymbol{C}=\boldsymbol{A}+(\boldsymbol{B}+\boldsymbol{C})$ 加法结合律

3. $r(\boldsymbol{A}+\boldsymbol{B})=r\boldsymbol{A}+r\boldsymbol{B}$ 标量乘法对矩阵加法的分配律

4. $(r+s)\boldsymbol{A}=r\boldsymbol{A}+s\boldsymbol{A}$ 矩阵乘法对标量加法的分配律

2.2 矩阵乘法

2.2.1 定义

如果$\boldsymbol{A}$是一个$m \times n$矩阵，$\boldsymbol{B}$是一个$n \times p$矩阵，那么，两者乘积$\boldsymbol{AB}$的结果是一个规模为$m \times p$的矩阵$\boldsymbol{C}$。矩阵$\boldsymbol{C}$中第i行、第j列的元素，由矩阵$\boldsymbol{A}$的第i个行向量与矩阵$\boldsymbol{B}$的第j个列向量的点积求得，即：

$$C_{ij} = \boldsymbol{A}_{i,*} \cdot \boldsymbol{B}_{*,j} \qquad (2.1)$$

要注意的是，为了使矩阵乘积$\boldsymbol{AB}$有意义，矩阵$\boldsymbol{A}$中的列数与矩阵$\boldsymbol{B}$中的行数必须相同。也就是说，矩阵$\boldsymbol{A}$中行向量的维数（可认为是分量的个数）与矩阵$\boldsymbol{B}$中列向量的维数要一致。如果二者的维数不同，那么式（2.1）中的点积运算没有意义。

例2. 3

设

$$\boldsymbol{A} = \begin{bmatrix} 1 & 5 \\ -2 & 3 \end{bmatrix} \text{和} \quad \boldsymbol{B} = \begin{bmatrix} 2 & -6 \\ 1 & 3 \\ -3 & 0 \end{bmatrix}$$

因为矩阵$\boldsymbol{A}$的行向量维数为2，矩阵$\boldsymbol{B}$的列向量维数为3，所以乘积$\boldsymbol{AB}$无定义。不妨这样想，由于2D向量不能与3D向量进行点积计算，因此，矩阵$\boldsymbol{A}$中的第一个行向量与矩阵$\boldsymbol{B}$中的第一个列向量也就无法开展

点积运算。

例2.4

设

$$\boldsymbol{A}=\begin{bmatrix}-1 & 5 & -4\\ 3 & 2 & 1\end{bmatrix} 与 \boldsymbol{B}=\begin{bmatrix}2 & 1 & 0\\ 0 & -2 & 1\\ -1 & 2 & 3\end{bmatrix}$$

由于矩阵$\boldsymbol{A}$的列数与矩阵$\boldsymbol{B}$的行数相同，可首先指出乘积$\boldsymbol{AB}$是有意义的（其结果是一个2×3矩阵）。根据式（2.1）可以得到：

$$\begin{aligned}\boldsymbol{AB}&=\begin{bmatrix}-1 & 5 & -4\\ 3 & 2 & 1\end{bmatrix}\begin{bmatrix}2 & 1 & 0\\ 0 & -2 & 1\\ -1 & 2 & 3\end{bmatrix}\\ &=\begin{bmatrix}(-1,5,-4)\cdot(2,0,-1) & (-1,5,-4)\cdot(1,-2,2) & (-1,5,-4)\cdot(0,1,3)\\ (3,2,1)\cdot(2,0,-1) & (3,2,1)\cdot(1,-2,2) & (3,2,1)\cdot(0,1,3)\end{bmatrix}\\ &=\begin{bmatrix}2 & -19 & -7\\ 5 & 1 & 5\end{bmatrix}\end{aligned}$$

我们还可以发现乘积$\boldsymbol{BA}$却没有意义，因为矩阵$\boldsymbol{B}$的列数不等于矩阵$\boldsymbol{A}$的行数。这表明，矩阵的乘法一般不满足交换律，即$\boldsymbol{AB}\neq\boldsymbol{BA}$。

2.2.2 向量与矩阵的乘法

考虑下列向量与矩阵的乘法运算：

$$\boldsymbol{uA}=[x,y,z]\begin{bmatrix}A_{11} & A_{12} & A_{13}\\ A_{21} & A_{22} & A_{23}\\ A_{31} & A_{32} & A_{33}\end{bmatrix}=[x,y,z]\begin{bmatrix}\uparrow & \uparrow & \uparrow\\ \boldsymbol{A}_{*,1} & \boldsymbol{A}_{*,2} & \boldsymbol{A}_{*,3}\\ \downarrow & \downarrow & \downarrow\end{bmatrix}$$

可以观察到：该例中，$\boldsymbol{uA}$的计算结果是一个规模为1×3的行向量。现在运用式（2.1）即可得到：

$$
\begin{aligned}
\boldsymbol{uA} &= [\boldsymbol{u}\cdot\boldsymbol{A}_{*,1} \quad \boldsymbol{u}\cdot\boldsymbol{A}_{*,2} \quad \boldsymbol{u}\cdot\boldsymbol{A}_{*,3}] \\
&= [xA_{11}+yA_{21}+zA_{31}, \quad xA_{12}+yA_{22}+zA_{32}, \quad xA_{13}+yA_{23}+zA_{33}] \\
&= [xA_{11}, xA_{12}, xA_{13}] + [yA_{21}, yA_{22}, yA_{23}] + [zA_{31}, zA_{32}, zA_{33}] \\
&= x\,[A_{11}, A_{12}, A_{13}] + y\,[A_{21}, A_{22}, A_{23}] + z\,[A_{31}, A_{32}, A_{33}] \\
&= x\boldsymbol{A}_{1,*} + y\boldsymbol{A}_{2,*} + z\boldsymbol{A}_{3,*}
\end{aligned}
$$

因此，

$$\boldsymbol{uA} = x\boldsymbol{A}_{1,*} + y\boldsymbol{A}_{2,*} + z\boldsymbol{A}_{3,*} \qquad (2.2)$$

式（2.2）实为一种线性组合（linear combination），这意味着向量与矩阵的乘积$\boldsymbol{uA}$就相当于：向量$\boldsymbol{u}$给定的标量系数x、y、z与矩阵$\boldsymbol{A}$中各行向量的线性组合。注意，尽管我们只展示了1×3行向量与3×3矩阵的乘法，但是这个结论却具有一般性。也就是说，对于一个$1\times n$行向量$\boldsymbol{u}$与一个$n\times m$矩阵$\boldsymbol{A}$，我们总可得到$\boldsymbol{u}$所给出的标量系数与$\boldsymbol{A}$中诸行向量的线性组合$\boldsymbol{uA}$：

$$[u_1, \cdots, u_n]\begin{bmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nm} \end{bmatrix} = u_1\boldsymbol{A}_{1,*} + \cdots + u_n\boldsymbol{A}_{n,*} \qquad (2.3)$$

2.2.3 结合律

矩阵的乘法运算具有一些很有用的代数性质。例如，矩阵乘法对矩阵加法的分配律$\boldsymbol{A}(\boldsymbol{B}+\boldsymbol{C}) = \boldsymbol{AB}+\boldsymbol{AC}$以及$(\boldsymbol{A}+\boldsymbol{B})\boldsymbol{C} = \boldsymbol{AC}+\boldsymbol{BC}$。除

此之外，我们还会不时地用到矩阵乘法的结合律，可借此来决定矩阵乘法的计算顺序：

$$(\boldsymbol{AB})\boldsymbol{C} = \boldsymbol{A}(\boldsymbol{BC})$$

2.3 转置矩阵

转置矩阵（transpose matrix）指的是将原矩阵的行与列进行互换所得到的新矩阵。所以，根据一个$m \times n$矩阵可得到一个规模为$n \times m$的转置矩阵。我们将矩阵$\boldsymbol{M}$的转置矩阵记作$\boldsymbol{M}^T$。

例2. 5

求出下列3个矩阵的转置矩阵：

$$\boldsymbol{A}=\begin{bmatrix}2 & -1 & 8\\ 3 & 6 & -4\end{bmatrix},\ \boldsymbol{B}=\begin{bmatrix}a & b & c\\ d & e & f\\ g & h & i\end{bmatrix},\ \boldsymbol{C}=\begin{bmatrix}1\\ 2\\ 3\\ 4\end{bmatrix}$$

上面提到，通过互换矩阵的行和列来求出转置矩阵，于是得到：

$$\boldsymbol{A}^{\mathrm{T}}=\begin{bmatrix}2 & 3\\ -1 & 6\\ 8 & -4\end{bmatrix},\ \boldsymbol{B}^{\mathrm{T}}=\begin{bmatrix}a & d & g\\ b & e & h\\ c & f & i\end{bmatrix},\ \boldsymbol{C}^{\mathrm{T}}=\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix}$$

转置矩阵具有下列实用性质：

1. $(\boldsymbol{A}+\boldsymbol{B})^T=\boldsymbol{A}^T+\boldsymbol{B}^T$

2. $(c\boldsymbol{A})^T=c\boldsymbol{A}^T$

3. $(\boldsymbol{AB})^T=\boldsymbol{B}^T\boldsymbol{A}^T$

4. $(\boldsymbol{A}^T)^T=\boldsymbol{A}$

5. $(\boldsymbol{A}^{-1})^T = (\boldsymbol{A}^T)^{-1}$

2.4 单位矩阵

单位矩阵（identity matrix）比较特殊，是一种主对角线上的元素均为1，其他元素都为0的方阵。

例如，下列依次是规模为2×2，3×3和4×4的单位矩阵：

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

单位矩阵是矩阵的乘法单位元（multiplicative identity）。即如果$\boldsymbol{A}$为$m \times n$矩阵，$\boldsymbol{B}$为$n \times p$矩阵，而$\boldsymbol{I}$为$n \times n$的单位矩阵，那么

$$\boldsymbol{AI} = \boldsymbol{A} \text{ 且 } \boldsymbol{IB} = \boldsymbol{B}$$

换句话说，任何矩阵与单位矩阵相乘，得到的依然是原矩阵。我们可以将单位矩阵看作是矩阵中的“数字1”。特别地，如果$\boldsymbol{M}$是一个方阵，那么它与单位矩阵的乘法满足交换律：

$$\boldsymbol{MI} = \boldsymbol{IM} = \boldsymbol{M}$$

例2.6

设$\boldsymbol{M} = \begin{bmatrix} 1 & 2 \\ 0 & 4 \end{bmatrix}$以及$\boldsymbol{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$，证明$\boldsymbol{MI} = \boldsymbol{IM} = \boldsymbol{M}$。

运用式（2.1）得：

$$\boldsymbol{MI} = \begin{bmatrix}1 & 2\\0 & 4\end{bmatrix}\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} = \begin{bmatrix}(1,2)\cdot(1,0) & (1,2)\cdot(0,1)\\(0,4)\cdot(1,0) & (0,4)\cdot(0,1)\end{bmatrix} = \begin{bmatrix}1 & 2\\0 & 4\end{bmatrix}$$

与

$$\boldsymbol{IM} = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}1 & 2\\0 & 4\end{bmatrix} = \begin{bmatrix}(1,0)\cdot(1,0) & (1,0)\cdot(2,4)\\(0,1)\cdot(1,0) & (0,1)\cdot(2,4)\end{bmatrix} = \begin{bmatrix}1 & 2\\0 & 4\end{bmatrix}$$

所以$\boldsymbol{MI} = \boldsymbol{IM} = \boldsymbol{M}$是正确的。

例2. 7

设$\boldsymbol{u}$ = [−1, 2]且$\boldsymbol{I} = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$。验证$\boldsymbol{uI} = \boldsymbol{u}$。

应用式（2.1），可得：

$$\boldsymbol{uI} = [-1,2]\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} = [(-1,\ 2)\cdot(1,\ 0),\ \ (-1,\ 2)\cdot(0,\ 1)] = [-1,\ 2]$$

另外可以看出，我们无法计算乘积$\boldsymbol{Iu}$，因为此矩阵乘法是无定义的。

2.5 矩阵的行列式

行列式是一种特殊的函数，它以一个方阵作为输入，并输出一个实数。方阵$\boldsymbol{A}$的行列式通常表示为$\det \boldsymbol{A}$。我们可以从几何的角度来解释行列式。行列式反映了在线性变换下，（n维多面体）体积变化的相关信息[2]。另外，行列式也应用于解线性方程组的克莱姆法则（Cramer’s Rule，亦称克莱默法则）。然而，我们在此学习行列式的主要目的是：利用它推导出求逆矩阵的公式（第2.7节的主题）。此外，行列式还可以用于证明：方阵$\boldsymbol{A}$是可逆的，当且仅当$\det \boldsymbol{A} \neq 0$。这个结论很实用，因为它为我们确认矩阵的可逆性提供了一种行之有效的计算工具。不过在定义行列式之前，我们先要介绍一下余子阵的概念。

2.5.1 余子阵

指定一个$n \times n$的矩阵$\boldsymbol{A}$，余子阵（minor matrix）[3]$\bar{\boldsymbol{A}}_{ij}$.即为从$\boldsymbol{A}$中去除第$i$行和第$j$列的$(n-1) \times (n-1)$矩阵。

例2.8

求出下列矩阵的余子阵$\bar{\boldsymbol{A}}_{11}$、$\bar{\boldsymbol{A}}_{22}$和$\bar{\boldsymbol{A}}_{13}$。

$$\boldsymbol{A} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

去除矩阵$\boldsymbol{A}$的第一行和第一列，得到$\bar{\boldsymbol{A}}_{11}$为：

$$\bar{\boldsymbol{A}}_{11} = \begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix}$$

去除矩阵$\boldsymbol{A}$的第二行和第二列，得到$\bar{\boldsymbol{A}}_{22}$为：

$$\bar{\boldsymbol{A}}_{22} = \begin{bmatrix} A_{11} & A_{13} \\ A_{31} & A_{33} \end{bmatrix}$$

去除矩阵$\boldsymbol{A}$的第一行和第三列，得到$\bar{\boldsymbol{A}}_{13}$为：

$$\bar{\boldsymbol{A}}_{13} = \begin{bmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}$$

2.5.2　行列式的定义

矩阵的行列式有一种递归定义。例如，一个4×4矩阵的行列式要根据3×3矩阵的行列式来定义，而3×3矩阵的行列式要靠2×2矩阵的行列式来定义，最后，2×2矩阵的行列式则依赖于1×1矩阵的行列式来定义（1×1矩阵$\boldsymbol{A} = [A_{11}]$的行列式被简单地定义为$\det[A_{11}] = A_{11}$）。

设$\boldsymbol{A}$为一个$n \times n$矩阵。那么，当$n > 1$时，我们定义：

$$\det \boldsymbol{A} = \sum_{j=1}^{n} A_{1j}(-1)^{1+j} \det \bar{\boldsymbol{A}}_{1j} \tag{2.4}$$

对照余子阵$\bar{\boldsymbol{A}}_{ij}$的定义可知，对于$2 \times 2$矩阵来说，其相应的行列式公式为：

$$\det\begin{bmatrix}A_{11} & A_{12}\\A_{21} & A_{22}\end{bmatrix}=A_{11}\det[A_{22}]-A_{12}\det[A_{21}]=A_{11}A_{22}-A_{12}A_{21}$$

对于3×3矩阵来说，其行列式计算公式为：

$$\begin{aligned}&\det\begin{bmatrix}A_{11} & A_{12} & A_{13}\\A_{21} & A_{22} & A_{23}\\A_{31} & A_{32} & A_{33}\end{bmatrix}\\&=A_{11}\det\begin{bmatrix}A_{22} & A_{23}\\A_{32} & A_{33}\end{bmatrix}-A_{12}\det\begin{bmatrix}A_{21} & A_{23}\\A_{31} & A_{33}\end{bmatrix}+A_{13}\det\begin{bmatrix}A_{21} & A_{22}\\A_{31} & A_{32}\end{bmatrix}\end{aligned}$$

对于4×4矩阵，其行列式计算公式为：

$$\begin{aligned}\det\begin{bmatrix}A_{11} & A_{12} & A_{13} & A_{14}\\A_{21} & A_{22} & A_{23} & A_{24}\\A_{31} & A_{32} & A_{33} & A_{34}\\A_{41} & A_{42} & A_{43} & A_{44}\end{bmatrix}&=A_{11}\det\begin{bmatrix}A_{22} & A_{23} & A_{24}\\A_{32} & A_{33} & A_{34}\\A_{42} & A_{43} & A_{44}\end{bmatrix}-A_{12}\det\begin{bmatrix}A_{21} & A_{23} & A_{24}\\A_{31} & A_{33} & A_{34}\\A_{41} & A_{43} & A_{44}\end{bmatrix}\\&+A_{13}\det\begin{bmatrix}A_{21} & A_{22} & A_{24}\\A_{31} & A_{32} & A_{34}\\A_{41} & A_{42} & A_{44}\end{bmatrix}-A_{14}\det\begin{bmatrix}A_{21} & A_{22} & A_{23}\\A_{31} & A_{32} & A_{33}\\A_{41} & A_{42} & A_{43}\end{bmatrix}\end{aligned}$$

在3D图形学中，主要使用4×4矩阵。因此，我们不再继续推导$n>4$的行列式公式。

例2.9

求矩阵$\boldsymbol{A}=\begin{bmatrix}2 & -5 & 3\\1 & 3 & 4\\-2 & 3 & 7\end{bmatrix}$的行列式。

我们有

$$\det \boldsymbol{A} = A_{11} \det \begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix} - A_{12} \det \begin{bmatrix} A_{21} & A_{23} \\ A_{31} & A_{33} \end{bmatrix} + A_{13} \det \begin{bmatrix} A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix}$$

$$\begin{aligned} \det \boldsymbol{A} &= 2 \det \begin{bmatrix} 3 & 4 \\ 3 & 7 \end{bmatrix} - (-5) \det \begin{bmatrix} 1 & 4 \\ -2 & 7 \end{bmatrix} + 3 \det \begin{bmatrix} 1 & 3 \\ -2 & 3 \end{bmatrix} \\ &= 2(3 \cdot 7 - 4 \cdot 3) + 5(1 \cdot 7 - 4 \cdot (-2)) + 3(1 \cdot 3 - 3 \cdot (-2)) \\ &= 2(9) + 5(15) + 3(9) \\ &= 18 + 75 + 27 \\ &= 120 \end{aligned}$$

2.6 伴随矩阵

设$\boldsymbol{A}$为一个$n \times n$矩阵。乘积$C_{ij} = (-1)^{i+j}\det\ \bar{\boldsymbol{A}}_{ij}$称为元素$\boldsymbol{A_{ij}}$的代数余子式（cofactor of A_{ij}）。如果为矩阵$\boldsymbol{A}$中的每个元素分别计算出C_{ij}，并将它置于矩阵$\boldsymbol{C_A}$中第i行、第j列的相应位置，那么将获得矩阵$\boldsymbol{A}$的代数余子式矩阵（cofactor matrix of $\boldsymbol{A}$）：

$$\boldsymbol{C}_A = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1n} \\ C_{21} & C_{22} & \cdots & C_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nn} \end{bmatrix}$$

若取矩阵$\boldsymbol{C_A}$的转置矩阵，将得到矩阵$\boldsymbol{A}$的伴随矩阵（adjoint matrix of $\boldsymbol{A}$），记作：

$$\boldsymbol{A}^* = \boldsymbol{C}_A^T \qquad (2.5)$$

在下一节中，我们将学习利用带有伴随矩阵的公式来计算逆矩阵。

2.7　逆矩阵

矩阵代数不存在除法运算的概念[4]，但是却另外定义了一种矩阵乘法的逆运算。下面总结了与矩阵逆运算有关的关键信息。

1．只有方阵才具有逆矩阵。因此，当提到逆矩阵时，我们便假设要处理的是一个方阵。

2．$n \times n$矩阵$\boldsymbol{M}$的逆也是一个$n \times n$矩阵，并表示为$\boldsymbol{M}^{-1}$。

3．不是每个方阵都有逆矩阵。存在逆矩阵的方阵称为可逆矩阵（invertible matrix），不存在逆矩阵的方阵称作奇异矩阵（singular matrix）。

4．可逆矩阵的逆矩阵是唯一的。

5．矩阵与其逆矩阵相乘将得到单位方阵：$\boldsymbol{M}\boldsymbol{M}^{-1} = \boldsymbol{M}^{-1}\boldsymbol{M} = \boldsymbol{I}$。可以发现，矩阵与其逆矩阵的乘法运算满足交换律。

另外，可以利用逆矩阵来解矩阵方程。例如，设矩阵方程$\boldsymbol{p}' = \boldsymbol{p}\boldsymbol{M}$，且已知$\boldsymbol{p}'$与$\boldsymbol{M}$，求$\boldsymbol{p}$。假设矩阵$\boldsymbol{M}$是可逆的（即存在$\boldsymbol{M}^{-1}$），我们就能解得$\boldsymbol{p}$。过程如下：

$$\boldsymbol{p}' = \boldsymbol{p}\boldsymbol{M}$$

$$\boldsymbol{p}'\boldsymbol{M}^{-1} = \boldsymbol{p}\boldsymbol{M}\boldsymbol{M}^{-1} \qquad \text{方程两端各乘以}\boldsymbol{M}^{-1}$$

$\boldsymbol{p}'\boldsymbol{M}^{-1}=\boldsymbol{p}\boldsymbol{I}$　　　　　　　根据可逆矩阵的定义，有$\boldsymbol{M}\boldsymbol{M}^{-1}=\boldsymbol{I}$

$\boldsymbol{p}'\boldsymbol{M}^{-1}=\boldsymbol{p}$　　　　　　　根据单位矩阵的定义，有$\boldsymbol{p}\boldsymbol{I}=\boldsymbol{p}$

在任何一本大学水平的线性代数教科书里，都可以找到求逆矩阵公式的推导过程，这里也就不再赘述了。此公式由原矩阵的伴随矩阵和行列式构成：

$$\boldsymbol{A}^{-1}=\frac{\boldsymbol{A}^{*}}{\det\boldsymbol{A}} \qquad (2.6)$$

例2.10

推导2×2矩阵$\boldsymbol{A}=\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}$的逆矩阵通式，并利用此式求出矩阵$\boldsymbol{M}=\begin{bmatrix}3 & 0\\ -1 & 2\end{bmatrix}$的逆矩阵。

已知

$$\det\boldsymbol{A}=A_{11}A_{22}-A_{12}A_{21}$$

$$\boldsymbol{C}_{A}=\begin{bmatrix}(-1)^{1+1}\det\bar{\boldsymbol{A}}_{11} & (-1)^{1+2}\det\bar{\boldsymbol{A}}_{12}\\ (-1)^{2+1}\det\bar{\boldsymbol{A}}_{21} & (-1)^{2+2}\det\bar{\boldsymbol{A}}_{22}\end{bmatrix}=\begin{bmatrix}A_{22} & -A_{21}\\ -A_{12} & A_{11}\end{bmatrix}$$

因此，

$$\boldsymbol{A}^{-1}=\frac{\boldsymbol{A}^{*}}{\det\ \boldsymbol{A}}=\frac{\boldsymbol{C}_{A}^{T}}{\det\ \boldsymbol{A}}=\frac{1}{A_{11}A_{22}-A_{12}A_{21}}\begin{bmatrix}A_{22} & -A_{12}\\ -A_{21} & A_{11}\end{bmatrix}$$

现在运用此公式来求矩阵$\boldsymbol{M}=\begin{bmatrix}3 & 0\\ -1 & 2\end{bmatrix}$的逆矩阵：

$$M^{-1} = \frac{1}{3 \cdot 2 - 0 \cdot (-1)} \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix} = \begin{bmatrix} 1/3 & 0 \\ 1/6 & 1/2 \end{bmatrix}$$

为了核实结果，我们来验证$MM^{-1} = M^{-1}M = I$：

$$\begin{bmatrix} 3 & 0 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} 1/3 & 0 \\ 1/6 & 1/2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1/3 & 0 \\ 1/6 & 1/2 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ -1 & 2 \end{bmatrix}$$

注意 Note

对于规模较小的矩阵（4×4及其以下规模的矩阵）来说，运用伴随矩阵的方法将得到不错的计算效率。但针对规模更大的矩阵而言，就要使用诸如高斯消元法（Gaussian elimination，也作高斯消去法）等其他手段。由于我们关注于3D计算机图形学中所涉及的具有特殊形式的矩阵，因此也就提前确定出了它们的求逆矩阵公式。这样一来，我们便无须在求常用的逆矩阵上浪费CPU资源了，继而也就极少会在代码中运用式（2.6）。

我们以下列“矩阵乘积的逆”这一实用的代数性质，为此节画上句号：

$$(AB)^{-1} = B^{-1}A^{-1}$$

该性质假设矩阵A与矩阵B都是可逆的，而且皆为同维方阵。为了证明$B^{-1}A^{-1}$是乘积AB的逆，我们必须证实$(AB)(B^{-1}A^{-1}) = I$以及

$(\boldsymbol{B}^{-1}\boldsymbol{A}^{-1})(\boldsymbol{A}\boldsymbol{B}) = \boldsymbol{I}$。证明过程如下：

$$(\boldsymbol{A}\boldsymbol{B})(\boldsymbol{B}^{-1}\boldsymbol{A}^{-1}) = \boldsymbol{A}(\boldsymbol{B}\boldsymbol{B}^{-1})\boldsymbol{A}^{-1} = \boldsymbol{A}\boldsymbol{I}\boldsymbol{A}^{-1} = \boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}$$

$$(\boldsymbol{B}^{-1}\boldsymbol{A}^{-1})(\boldsymbol{A}\boldsymbol{B}) = \boldsymbol{B}^{-1}(\boldsymbol{A}^{-1}\boldsymbol{A})\boldsymbol{B} = \boldsymbol{B}^{-1}\boldsymbol{I}\boldsymbol{B} = \boldsymbol{B}^{-1}\boldsymbol{B} = \boldsymbol{I}$$

2.8 用DirectXMath库处理矩阵

为了对点与向量进行变换，就要借助1×4行向量以及4×4矩阵。相关原因将在下一章中细述。目前，我们只需把注意力集中在DirectXMath库中常用于表示4×4矩阵的数据类型。

2.8.1 矩阵类型

DirectXMath以定义在DirectXMath.h头文件中的XMMATRIX类来表示4×4矩阵（为了叙述清晰起见，这里进行了若干细节上的调整）：

```
#if (defined(_M_IX86) || defined(_M_X64) || defined(_M_ARM)) &&
defined(_XM_NO_INTRINSICS_)
struct XMMATRIX
#else
__declspec(align(16)) struct XMMATRIX
#endif
{
  // 利用4个XMVECTOR来表示矩阵，借此使用SIMD技术
  XMVECTOR r[4];

  XMMATRIX() {}

  // 通过指定4个行向量来初始化矩阵
  XMMATRIX(FXMVECTOR R0, FXMVECTOR R1, FXMVECTOR R2, CXMVECTOR R3)
    { r[0] = R0; r[1] = R1; r[2] = R2; r[3] = R3; }

  // 通过指定16个矩阵元素来初始化矩阵
  XMMATRIX(float m00, float m01, float m02, float m03,
       float m10, float m11, float m12, float m13,
       float m20, float m21, float m22, float m23,
       float m30, float m31, float m32, float m33);

  // 通过含有16个浮点数元素的数组来初始化矩阵
  explicit XMMATRIX(_In_reads_(16) const float *pArray);
```

```
  XMMATRIX&  operator= (const XMMATRIX& M)
    { r[0] = M.r[0]; r[1] = M.r[1]; r[2] = M.r[2]; r[3] = M.r[3];
    return *this; }

  XMMATRIX  operator+ () const { return *this; }
  XMMATRIX  operator- () const;

  XMMATRIX&  XM_CALLCONV   operator+= (FXMMATRIX M);
  XMMATRIX&  XM_CALLCONV   operator-= (FXMMATRIX M);
  XMMATRIX&  XM_CALLCONV   operator*= (FXMMATRIX M);
  XMMATRIX&  operator*= (float S);
  XMMATRIX&  operator/= (float S);

  XMMATRIX  XM_CALLCONV   operator+ (FXMMATRIX M) const;
  XMMATRIX  XM_CALLCONV   operator- (FXMMATRIX M) const;
  XMMATRIX  XM_CALLCONV   operator* (FXMMATRIX M) const;
  XMMATRIX  operator* (float S) const;
  XMMATRIX  operator/ (float S) const;

  friend XMMATRIX   XM_CALLCONV   operator* (float S, FXMMATRIX M);
};
```

综上所述，XMMATRIX由4个XMVECTOR实例所构成，并借此来使用SIMD技术。此外，XMMATRIX类还为矩阵计算提供了多种重载运算符。

除了各种构造方法之外，还可以使用XMMatrixSet函数来创建XMMATRIX实例：

```
XMMATRIX XM_CALLCONV XMMatrixSet(
  float m00, float m01, float m02, float m03,
  float m10, float m11, float m12, float m13,
  float m20, float m21, float m22, float m23,
  float m30, float m31, float m32, float m33);
```

就像通过XMFLOAT2 (2D)，XMFLOAT3 (3D)和XMFLOAT4 (4D)来存储类中不同维度的向量一样，DirectXMath文档也建议我们用XMFLOAT4X4来存储类中的矩阵类型数据成员。

```
struct XMFLOAT4X4
{
  union
  {
    struct
    {
      float _11, _12, _13, _14;
      float _21, _22, _23, _24;
      float _31, _32, _33, _34;
      float _41, _42, _43, _44;
    };
    float m[4][4];
  };

  XMFLOAT4X4() {}
  XMFLOAT4X4(float m00, float m01, float m02, float m03,
             float m10, float m11, float m12, float m13,
             float m20, float m21, float m22, float m23,
             float m30, float m31, float m32, float m33);
  explicit XMFLOAT4X4(_In_reads_(16) const float *pArray);

  float    operator() (size_t Row, size_t Column) const { return m[Row][Co
lumn]; }
  float&   operator() (size_t Row, size_t Column) { return m[Row][Column];
 }

  XMFLOAT4X4& operator=(const XMFLOAT4X4& Float4x4);
};
```

通过下列方法将数据从XMFLOAT4X4内加载到XMMATRIX中：

```
inline XMMATRIX XM_CALLCONV
XMLoadFloat4x4(const XMFLOAT4X4* pSource);
```

通过下列方法将数据从XMMATRIX内存储到XMFLOAT4X4中：

```
inline void XM_CALLCONV
XMStoreFloat4x4(XMFLOAT4X4* pDestination, FXMMATRIX M);
```

2.8.2 矩阵函数

DirectXMath库包含了下列与矩阵相关的实用函数：

```
XMMATRIX XM_CALLCONV XMMatrixIdentity();   // 返回单位矩阵I

bool XM_CALLCONV XMMatrixIsIdentity(       // 如果M是单位矩阵则返回true
    FXMMATRIX M);                          // 输入矩阵M

XMMATRIX XM_CALLCONV XMMatrixMultiply(     // 返回矩阵乘积AB
    FXMMATRIX A,                           // 输入矩阵A
    CXMMATRIX B);                          // 输入矩阵B

XMMATRIX XM_CALLCONV XMMatrixTranspose(    // 返回M^T
    FXMMATRIX M);                          // 输入矩阵M

XMVECTOR XM_CALLCONV XMMatrixDeterminant(  // 返回(det M, det M, det M, de
t M)
    FXMMATRIX M);                          // 输入矩阵M

XMMATRIX XM_CALLCONV XMMatrixInverse(      // 返回M^-1
    XMVECTOR* pDeterminant,                // 输入(det M, det M, det M, de
t M)
    FXMMATRIX M);                          // 输入矩阵M
```

在声明具有XMMATRIX参数的函数时，除了要注意1个XMMATRIX应计作4个XMVECTOR参数这一点之外，其他的规则与传入XMVECTOR类型的参数时（见1.6.3节）相一致。假设传入函数的FXMVECTOR参数不超过两个，则第一个XMMATRIX参数应当为FXMMATRIX类型，其余的XMMATRIX参数均应为CXMMATRIX类型。下面的代码展示了在32位Windows平台和编译器（编译器需支持__fastcall以及新增的__vectorcall调用约定）的环境下，这些类型的定义：

```
// 在32位的Windows系统上，__fastcall调用约定通过寄存器传递前3个XMVECTOR参数，
其余的
//参数则存在堆栈上
typedef const XMMATRIX& FXMMATRIX;
typedef const XMMATRIX& CXMMATRIX;

// 在32位的Windows系统上，__vectorcall调用约定通过寄存器传递前 6个 XMVECTOR实
```

```
参，其余的
// 参数则存在堆栈上
typedef const XMMATRIX FXMMATRIX;
typedef const XMMATRIX& CXMMATRIX;
```

可以看出，在32位Windows操作系统上的`__fastcall`调用约定中，`XMMATRIX`类型的参数是不能传至SSE/SSE2寄存器的，因为这些寄存器此时只支持3个`XMVECTOR`参数传入。而`XMMATRIX`参数却是由4个`XMVECTOR`构成，所以矩阵类型的数据只能通过堆栈来加以引用。至于这些类型在其他平台上的定义详情，可见DirectXMath文档[DirectXMath]中“Library Internals”（库的内部细节）下的“Calling Conventions”（调用约定）部分。构造函数方法对于这些规则来说是一个特例。[DirectXMath]建议用户总是在构造函数中采用`CXMMATRIX`类型来获取`XMMATRIX`参数，而且对于构造函数也不要使用`XM_CALLCONV`约定注解。

2.8.3　DirectXMath矩阵示例程序

下列代码提供了一些`XMMATRIX`类的使用范例，其中包括了上一小节中介绍的大多数函数。

```
#include <windows.h> // 为了使XMVerifyCPUSupport函数返回正确值
#include <DirectXMath.h>
#include <DirectXPackedVector.h>
#include <iostream>
using namespace std;
using namespace DirectX;
using namespace DirectX::PackedVector;

// 重载"<<"运算符，这样就可以利用cout输出XMVECTOR和XMMATRIX对象
ostream& XM_CALLCONV operator << (ostream& os, FXMVECTOR v)
```

```
{
  XMFLOAT4 dest;
  XMStoreFloat4(&dest, v);

  os << "(" << dest.x << ", " << dest.y << ", " << dest.z << ", " << dest.
w << ")";
  return os;
}

ostream& XM_CALLCONV operator << (ostream& os, FXMMATRIX m)
{
  for (int i = 0; i < 4; ++i)
  {
    os << XMVectorGetX(m.r[i]) << "\t";
    os << XMVectorGetY(m.r[i]) << "\t";
    os << XMVectorGetZ(m.r[i]) << "\t";
    os << XMVectorGetW(m.r[i]);
    os << endl;
  }
  return os;
}

int main()
{
  // 检查是否支持SSE2指令集 (Pentium4, AMD K8及其后续版本的处理器)
  if (!XMVerifyCPUSupport())
  {
    cout << "directx math not supported" << endl;
    return 0;
  }

  XMMATRIX A(1.0f, 0.0f, 0.0f, 0.0f,
             0.0f, 2.0f, 0.0f, 0.0f,
             0.0f, 0.0f, 4.0f, 0.0f,
             1.0f, 2.0f, 3.0f, 1.0f);

  XMMATRIX B = XMMatrixIdentity();
  XMMATRIX C = A * B;

  XMMATRIX D = XMMatrixTranspose(A);

  XMVECTOR det = XMMatrixDeterminant(A);
  XMMATRIX E = XMMatrixInverse(&det, A);

  XMMATRIX F = A * E;

  cout << "A = " << endl << A << endl;
```

```
    cout << "B = " << endl << B << endl;
    cout << "C = A*B = " << endl << C << endl;
    cout << "D = transpose(A) = " << endl << D << endl;
    cout << "det = determinant(A) = " << det << endl << endl;
    cout << "E = inverse(A) = " << endl << E << endl;
    cout << "F = A*E = " << endl << F << endl;

    return 0;
}
```

上述范例程序的输出结果如图2.1所示。

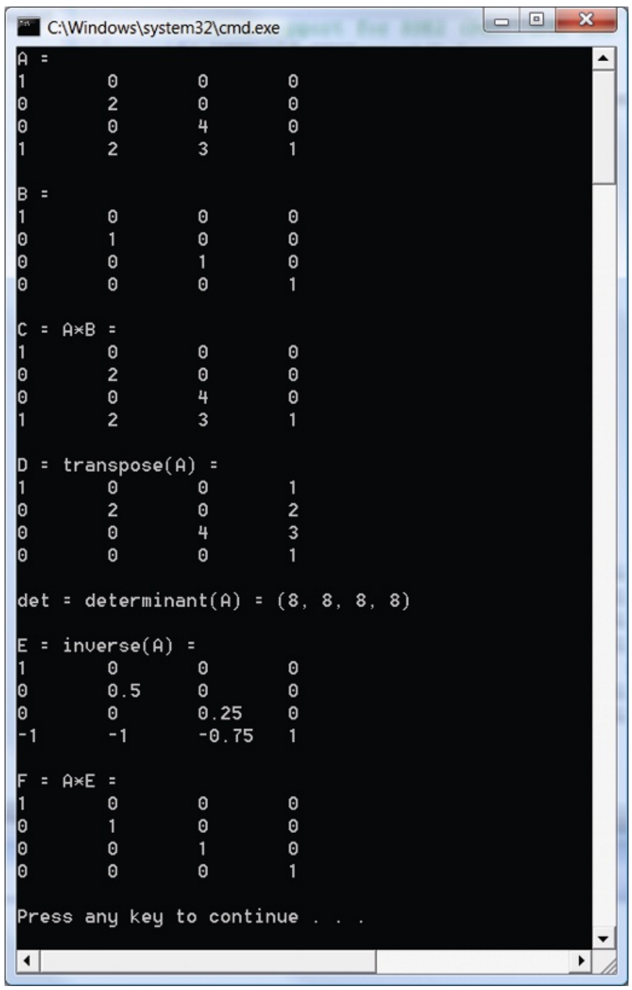


图2.1　范例程序输出的结果

2.9 小结

1．$m \times n$矩阵$\boldsymbol{M}$是一个由m行n列实数所构成的矩形阵列。两个同维矩阵相等，当且仅当它们对应的元素分别相等。两个同维矩阵的加法运算，由这两个矩阵对应的元素相加来实现。标量与矩阵的乘法运算是将标量与矩阵中的每个元素分别相乘。

2．如果$\boldsymbol{A}$是一个$m \times n$矩阵，且$\boldsymbol{B}$为一个$n \times p$矩阵，那么两者乘积$\boldsymbol{AB}$的结果是一个规模为$m \times p$的矩阵$\boldsymbol{C}$。矩阵$\boldsymbol{C}$中第i行、第j列的元素，由矩阵$\boldsymbol{A}$中的第i个行向量与矩阵$\boldsymbol{B}$中的第j个列向量进行点积运算得出，即$C_{ij} = \boldsymbol{A}_{i,*} \cdot \boldsymbol{B}_{*,j}$。

3．矩阵乘法不满足交换律（即一般来说，$\boldsymbol{AB} \neq \boldsymbol{BA}$），但是却满足结合律$(\boldsymbol{AB})\boldsymbol{C} = \boldsymbol{A}(\boldsymbol{BC})$。

4．转置矩阵由原矩阵互换行与列来求得。所以，$m \times n$矩阵的转置矩阵为$n \times m$矩阵。我们将矩阵$\boldsymbol{M}$的转置矩阵表示为$\boldsymbol{M}^T$。

5．单位矩阵是一种除主对角线上的元素为1外，其他元素均为0的方阵。

6．行列式$\det \boldsymbol{A}$是一种特殊的函数，向它传入一个方阵便会计算出一个对应的实数。方阵$\boldsymbol{A}$是可逆的，当且仅当$\det \boldsymbol{A} \neq 0$。行列式常常用于计算逆矩阵。

7．矩阵与其逆矩阵的乘积结果为单位矩阵，即 $\boldsymbol{M}\boldsymbol{M}^{-1}=\boldsymbol{M}^{-1}\boldsymbol{M}=\boldsymbol{I}$。如果一个矩阵是可逆的，则此矩阵的逆矩阵是唯一的。只有方阵才可能有逆矩阵，即便是方阵也未必可逆。逆矩阵可由公式$\boldsymbol{A}^{-1}=\boldsymbol{A}^{*}/\det\boldsymbol{A}$来计算，其中$\boldsymbol{A}^{*}$是伴随矩阵（即矩阵$\boldsymbol{A}$的代数余子式矩阵的转置矩阵）。

8．我们在编写代码时，用DirectXMath中的XMMATRIX类型来表示4×4矩阵，以此来发挥SIMD技术高效的运算能力。但对于类中的数据成员，我们则要以XMFLOAT4X4类型来加以表示，并通过加载（XMLoadFloat4x4）和存储（XMStoreFloat4x4）方法，使数据在XMMATRIX类型与XMFLOAT4X4类型之间互相转换。XMMATRIX类重载了一些算数运算符，使矩阵可以实现加法运算、减法运算、矩阵乘法运算和标量乘法运算。此外，DirectXMath库还提供了下列实用的矩阵函数，用于计算单位矩阵、矩阵乘积、转置矩阵、行列式以及逆矩阵：

```
XMMATRIX XM_CALLCONV XMMatrixIdentity();
XMMATRIX XM_CALLCONV XMMatrixMultiply(FXMMATRIX A, CXMMATRIX B);
XMMATRIX XM_CALLCONV XMMatrixTranspose(FXMMATRIX M);
XMVECTOR XM_CALLCONV XMMatrixDeterminant(FXMMATRIX M);
XMMATRIX XM_CALLCONV XMMatrixInverse(XMVECTOR* pDeterminant,
  FXMMATRIX M);
```

2.10　练习

1．求解下列矩阵方程中的矩阵$\boldsymbol{X}$：$3\left(\begin{bmatrix}-2 & 0\\ 1 & 3\end{bmatrix} - 2\boldsymbol{X}\right) = 2\begin{bmatrix}-2 & 0\\ 1 & 3\end{bmatrix}$。

2．计算下列矩阵的乘积：

（a）$\begin{bmatrix}-2 & 0 & 3\\ 4 & 1 & -1\end{bmatrix}\begin{bmatrix}2 & -1\\ 0 & 6\\ 2 & -3\end{bmatrix}$

（b）$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}\begin{bmatrix}-2 & 0\\ 1 & 1\end{bmatrix}$

（c）$\begin{bmatrix}2 & 0 & 2\\ 0 & -1 & -3\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1\\ 2\\ 1\end{bmatrix}$

3．计算下列矩阵的转置矩阵：

（a）$[1,\quad 2,\quad 3]$

（b）$\begin{bmatrix}x & y\\ z & w\end{bmatrix}$

（c）$\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\\ 7 & 8\end{bmatrix}$

4．将下列线性组合写作向量与矩阵乘积的形式：

（a）$\boldsymbol{v} = 2(1, 2, 3) - 4(-5, 0, -1) + 3(2, -2, 3)$

（b）$\boldsymbol{v} = 3(2, -4) + 2(1, 4) - 1(-2, -3) + 5(1, 1)$

5．证明

$$\boldsymbol{AB} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix} = \begin{bmatrix} \leftarrow \boldsymbol{A}_{1,*}\boldsymbol{B} \rightarrow \\ \leftarrow \boldsymbol{A}_{2,*}\boldsymbol{B} \rightarrow \\ \leftarrow \boldsymbol{A}_{3,*}\boldsymbol{B} \rightarrow \end{bmatrix}$$

6．证明

$$\boldsymbol{Au} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = x\boldsymbol{A}_{*,1} + y\boldsymbol{A}_{*,2} + z\boldsymbol{A}_{*,3}$$

7．证明向量的叉积可以用矩阵的乘积来表示：

$$\boldsymbol{u} \times \boldsymbol{v} = [v_x v_y v_z] \begin{bmatrix} 0 & u_z & -u_y \\ -u_z & 0 & u_x \\ u_y & -u_x & 0 \end{bmatrix}$$

8．设矩阵$\boldsymbol{A} = \begin{bmatrix} 2 & 0 & 1 \\ 0 & -1 & -3 \\ 0 & 0 & 1 \end{bmatrix}$，那么，请问矩阵 $\boldsymbol{B} = \begin{bmatrix} 1/2 & 0 & -1/2 \\ 0 & -1 & -3 \\ 0 & 0 & 1 \end{bmatrix}$是$\boldsymbol{A}$的逆矩阵吗？

9．设矩阵$\boldsymbol{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$，那么，请问矩阵$\boldsymbol{B} = \begin{bmatrix} -2 & 1 \\ 3/2 & 1/2 \end{bmatrix}$是$\boldsymbol{A}$的逆矩阵吗？

10．求下列矩阵的行列式：

$$\begin{bmatrix}21 & -4\\10 & 7\end{bmatrix} \qquad \begin{bmatrix}2 & 0 & 0\\0 & 3 & 0\\0 & 0 & 7\end{bmatrix}$$

11．求下列矩阵的逆矩阵：

$$\begin{bmatrix}21 & -4\\10 & 7\end{bmatrix} \qquad \begin{bmatrix}2 & 0 & 0\\0 & 3 & 0\\0 & 0 & 7\end{bmatrix}$$

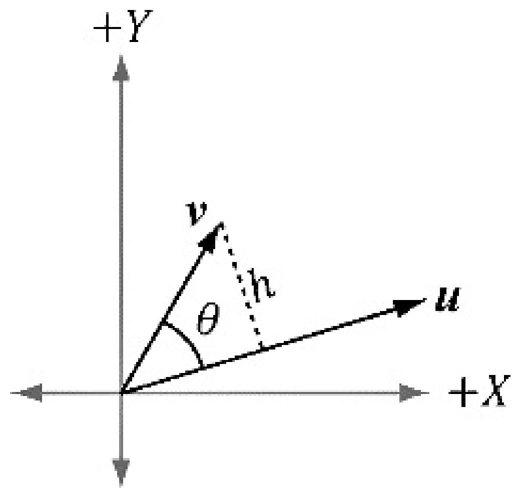
12．下列矩阵是可逆矩阵吗？

$$\begin{bmatrix}1 & 2 & 3\\0 & 4 & 5\\0 & 0 & 0\end{bmatrix}$$

13．假设矩阵$\boldsymbol{A}$是可逆矩阵，证明$(\boldsymbol{A}^{-1})^T = (\boldsymbol{A}^T)^{-1}$。

14．所有的线性代数书籍都会证明$\det(\boldsymbol{AB}) = \det\ \boldsymbol{A} \cdot \det\ \boldsymbol{B}$这一性质。设$\boldsymbol{A}$和$\boldsymbol{B}$皆为$n \times n$矩阵，并假设$\boldsymbol{A}$是可逆的，试根据$\det \boldsymbol{I} = 1$与上述性质来证明$\det\ \boldsymbol{A}^{-1} = \dfrac{1}{\det\ \boldsymbol{A}}$。

15．证明2D矩阵$\begin{bmatrix}u_x & u_y\\v_x & v_y\end{bmatrix}$的行列式得到的是：由向量$\boldsymbol{u} = (u_x, u_y)$与向量$\boldsymbol{v} = (v_x, v_y)$张成的平行四边形的有向面积。如果向量$\boldsymbol{u}$以逆时针方向旋转角$\boldsymbol{\theta} \in (0, \pi)$能与向量$\boldsymbol{v}$重合，则结果为正，否则为负。



16．求由下列向量张成的平行四边形面积：

（a）$\boldsymbol{u}=(3,0)\boldsymbol{v}=(1,1)$

（b）$\boldsymbol{u}=(-1,-1)\boldsymbol{v}=(0,1)$

17．设$\boldsymbol{A}=\begin{bmatrix}A_{11} & A_{12}\\A_{21} & A_{22}\end{bmatrix}$ $\boldsymbol{B}=\begin{bmatrix}B_{11} & B_{12}\\B_{21} & B_{22}\end{bmatrix}$，且$\boldsymbol{C}=\begin{bmatrix}C_{11} & C_{12}\\C_{21} & C_{22}\end{bmatrix}$。证明$\boldsymbol{A}(\boldsymbol{BC})=(\boldsymbol{AB})\boldsymbol{C}$。这个结论说明了$2\times 2$矩阵之间的乘法运算满足结合律。（事实上，只要矩阵的乘法有意义，任意规模的矩阵乘法都满足结合律。）

18．编写一个计算机程序，使之在不借助DirectXMath库的情况下（仅用C++中的二维数组（array of arrays））就可以计算$m\times n$矩阵的转置矩阵。

19．编写一个计算机程序，在不使用DirectXMath库的情况下（仅用C++中的二维数组），使它可以计算出4×4矩阵的行列式及其逆矩阵。

[1]　准确来讲，矩阵中的元素并非仅为实数。

[2]　这个定义并不十分准确。例如，在二维情况的线性变换下，二阶行列式反映的是平行四边形有向面积的变化。参见本章练习15。

[3]　这一小节中的部分数学术语在不同的文献中会有些差别，此处以常见文献中的译法为主。读者应以具体的定义为准。

[4]　可参见22.2.6节。

第3章　变换

通过将一系列三角形拼接在一起即可近似地表示物体的外表面，我们借助这种几何方式来描述3D空间内的物体。但仅有静止物体的场景是索然无味的，所以我们将把兴致放在探求几何体变换的方法之上，如平移、旋转和缩放这几种常见的几何变换。在本章中，我们将推导用于在3D空间里对点和向量进行变换的矩阵方程。

学习目标：

1．理解如何用矩阵表示线性变换和仿射变换。

2．学习对几何体进行缩放、旋转和平移的坐标变换。

3．根据矩阵之间的乘法运算性质，将多个变换矩阵合并为一个单独的净变换矩阵。

4．找寻不同坐标系之间的坐标转换方法，并利用矩阵来表示此坐标变换。

5．熟悉DirectXMath库专为构建变换矩阵所提供的相关函数。

3.1　线性变换

3.1.1　定义

先来研究一下数学函数$\tau(\boldsymbol{v}) = \tau(x, y, z) = (x', y', z')$。此函数的输入和输出都是3D向量。我们称$\tau$为线性变换（linear transformation），当且仅当此函数具有下列性质：

$$\tau(\boldsymbol{u} + \boldsymbol{v}) = \tau(\boldsymbol{u}) + \tau(\boldsymbol{v})$$

$$\tau(k\boldsymbol{u}) = k\tau(\boldsymbol{u}) \qquad (3.1)$$

其中，$\boldsymbol{u} = (u_x, u_y, u_z)$和$\boldsymbol{v} = (v_x, v_y, v_z)$是任意3D向量，$k$为一个标量。

注意　Note

非3D向量亦可作为线性变换的输入和输出，但是在有关3D图形学相关的书籍中往往不会讨论这种一般情况。

例3.1

定义函数$\tau(x, y, z) = (x^2, y^2, z^2)$，例如$\tau(1, 2, 3) = (1, 4, 9)$。这个函数是

非线性函数，因为当$k=2$且$\boldsymbol{u}=(1,2,3)$时，有

$$\tau(k\boldsymbol{u})=\tau(2,4,6)=(4,16,36)$$

但

$$k\tau(\boldsymbol{u})=2(1,4,9)=(2,8,18)$$

因此，该函数不满足式（3.1）中的第二条性质。

如果τ是线性函数，那么有

$$\begin{aligned}&\tau(a\boldsymbol{u}+b\boldsymbol{v}+c\boldsymbol{w})=\tau(a\boldsymbol{u}+(b\boldsymbol{v}+c\boldsymbol{w}))\\&=a\tau(\boldsymbol{u})+\tau(b\boldsymbol{v}+c\boldsymbol{w})\\&=a\tau(\boldsymbol{u})+b\tau(\boldsymbol{v})+c\tau(\boldsymbol{w})\end{aligned} \quad（3.2）$$

我们在下一小节中将用到这个结论。

3.1.2 矩阵表示法

设$\boldsymbol{u}=(x,y,z)$，我们也可以将它写作

$$\boldsymbol{u}=(x,y,z)=x\boldsymbol{i}+y\boldsymbol{j}+z\boldsymbol{k}=x(1,0,0)+y(0,1,0)+z(0,0,1)$$

$\boldsymbol{i}=(1,0,0),\boldsymbol{j}=(0,1,0)$和$\boldsymbol{k}=(0,0,1)$分别表示位于当前坐标轴正方向上的3个单位向量，我们称之为$\mathbb{R}^3$（$\mathbb{R}^3$表示所有3D坐标向量(x, y, z)的集合）的标准基向量（standard basis vector）。现假设τ是一种线性变换，根据它的线性性质（即式（3.2）），能够得到

$$\tau(\boldsymbol{u})=\tau(x\boldsymbol{i}+y\boldsymbol{j}+z\boldsymbol{k})=x\tau(\boldsymbol{i})+y\tau(\boldsymbol{j})+z\tau(\boldsymbol{k}) \quad（3.3）$$

可以看出，这个公式其实就是我们在上一章中学到的线性组合，可将其表示为向量与矩阵的乘积。根据式（2.2），我们可以将式（3.3）改写作

$$
\begin{aligned}
\tau(\boldsymbol{u}) &= x\tau(\boldsymbol{i}) + y\tau(\boldsymbol{j}) + z\tau(\boldsymbol{k}) \\
&= \boldsymbol{uA} = [x,\ y,\ z]\begin{bmatrix} \leftarrow \tau(\boldsymbol{i}) \rightarrow \\ \leftarrow \tau(\boldsymbol{j}) \rightarrow \\ \leftarrow \tau(\boldsymbol{k}) \rightarrow \end{bmatrix} = [x,\ y,\ z]\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}
\end{aligned} \quad (3.4)
$$

其中$\tau(\boldsymbol{i}) = (A_{11}, A_{12}, A_{13})$，$\tau(\boldsymbol{j}) = (A_{21}, A_{22}, A_{23})$且$\tau(\boldsymbol{k}) = (A_{31}, A_{32}, A_{33})$。我们称矩阵$\boldsymbol{A}$是线性变换$\tau$的矩阵表示法。

3.1.3 缩放

缩放（scaling，也有译作比例变换）是指改变物体的大小，其变换效果如图3.1所示。

我们把缩放变换定义为

$$S(x, y, z) = (s_x x, s_y y, s_z z)$$

此变换将相对于当前坐标系中的原点，令向量在x、y、z轴上分别以系数s_x、s_y、s_z进行缩放。下面我们来证明S其实就是一种线性变换：

$$S(\boldsymbol{u} + \boldsymbol{v}) = (s_x(u_x + v_x), s_y(u_y + v_y), s_z(u_z + v_z))$$

$$= (s_x u_x + s_x v_x, s_y u_y + s_y v_y, s_z u_z + s_z v_z)$$

$$= (s_x u_x, s_y u_y, s_z u_z) + (s_x v_x, s_y v_y, s_z v_z)$$

$$= S(\boldsymbol{u}) + S(\boldsymbol{v})$$

$$S(k\boldsymbol{u}) = (s_x k u_x, s_y k u_y, s_z k u_z)$$

$$= k(s_x u_x, s_y u_y, s_z u_z)$$

$$= kS(\boldsymbol{u})$$

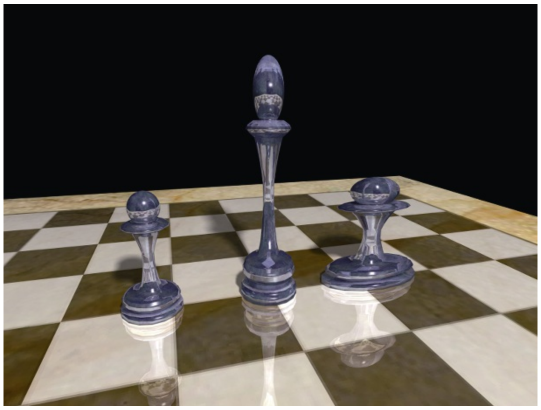


图3.1　左侧的兵是未经变换的原始物体，中间的兵是把原始的兵在y轴方向伸长2倍后“增高”的效果，右侧的兵是将原始的兵在x轴方向拃宽2倍后“增肥”的效果

因此，缩放变换S满足式（3.1）中的所有性质。这就是说，S是线性变换并存在一种矩阵表示法。为了求出S的矩阵表示，我们只需像式（3.3）那样，把每一个标准基向量依次代入S，再将得到的向量作为矩阵的行向量（计算过程如式（3.4））。

$$\begin{aligned}S(\boldsymbol{i}) &= (s_x \cdot 1,\ s_y \cdot 0,\ s_z \cdot 0) = (s_x,\ 0,\ 0)\\ S(\boldsymbol{j}) &= (s_x \cdot 0,\ s_y \cdot 1,\ s_z \cdot 0) = (0,\ s_y,\ 0)\\ S(\boldsymbol{k}) &= (s_x \cdot 0,\ s_y \cdot 0,\ s_z \cdot 1) = (0,\ 0,\ s_z)\end{aligned}$$

这样就得到了缩放变换S的矩阵表示

$$\boldsymbol{S}=\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

我们称此矩阵为缩放矩阵（scaling matrix，亦有译为比例变换矩阵）。

而其对应的逆矩阵则为

$$\boldsymbol{S}^{-1}=\begin{bmatrix} 1/s_x & 0 & 0 \\ 0 & 1/s_y & 0 \\ 0 & 0 & 1/s_z \end{bmatrix}$$

例3.2

假设定义了一个最小点坐标为(−4, −4, 0)和最大点坐标为(4, 4, 0)的正方形。现欲将此正方形在x轴方向上缩小50%，在y轴方向上放大2.0倍，但在z轴方向上保持不变。其对应的缩放矩阵为

$$\boldsymbol{S}=\begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

此时，若要对该正方形进行缩放（变换），只需将其最小点、最大点坐标分别与缩放矩阵相乘即可：

$$[-4,\ -4,\ 0]\begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}=[-2,\ -8,\ 0] \qquad [4,\ 4,\ 0]\begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}=[2,\ 8,\ 0]$$

变换的效果如图3.2所示。

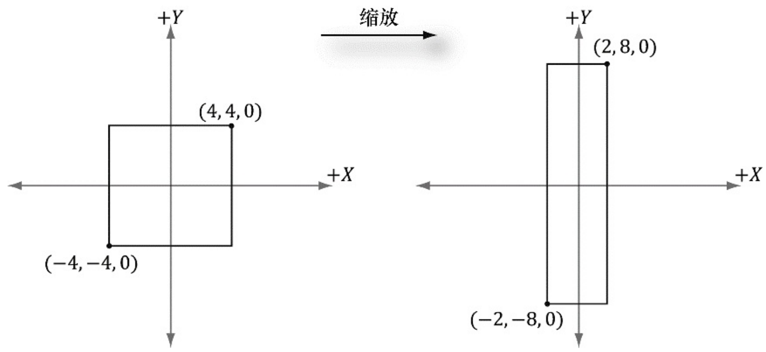


图3.2　将例3.2中的图像在x轴方向缩小为起始的50%、在y轴方向放大为起始的两倍后的效果。注意，在沿z轴负方向观察的时候，由于z = 0，所以几何体看上去是2D效果[1]

3.1.4　旋转

在本节中，我们将用数学的方式来描述令向量$\boldsymbol{v}$绕轴$\boldsymbol{n}$以角$\boldsymbol{\theta}$进行旋转，此过程如图3.3所示。注意，在沿$\boldsymbol{n}$轴从上至下俯瞰时，我们按顺时针方向来测量角$\boldsymbol{\theta}$，并且假设$||\boldsymbol{n}|| = 1$。

首先，将向量$\boldsymbol{v}$分解为两部分：一部分平行于$\boldsymbol{n}$，另一部分正交于$\boldsymbol{n}$。平行于$\boldsymbol{n}$的部分即为$\mathrm{proj}_{\boldsymbol{n}}(\boldsymbol{v})$（参见第1章中的例1.5），正交于$\boldsymbol{n}$的部分则是$\boldsymbol{v}_{\perp} = \mathrm{perp}_n(\boldsymbol{v}) = \boldsymbol{v} - \mathrm{proj}_n(\boldsymbol{v})$（同见例1.5，由于$\boldsymbol{n}$是单位向量，我们就可以得到$\mathrm{proj}_n(\boldsymbol{v}) = (\boldsymbol{n} \cdot \boldsymbol{v})\boldsymbol{n}$）。观察示意图能够发现这样一个关键信息：平行于$\boldsymbol{n}$的部分$\mathrm{proj}_{\boldsymbol{n}}(\boldsymbol{v})$在旋转时是保持不变的。因此，我们只需考虑怎样旋转与$\boldsymbol{n}$正交的部分。根据图3.3可知，旋转向量$R_n(\boldsymbol{v}) = \mathrm{proj}_n(\boldsymbol{v}) + R_n(\boldsymbol{v}_{\perp})$。

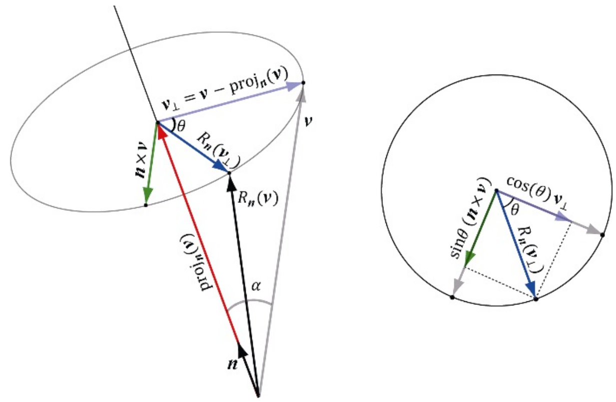


图3.3　令向量绕轴$\boldsymbol{n}$旋转的几何关系示意图

为了求出$R_n(\boldsymbol{v}_\perp)$，我们在旋转平面内建立一个2D坐标系，并将$\boldsymbol{v}_\perp$作为一个参考向量。通过计算叉积$\boldsymbol{n}\times\boldsymbol{v}$来获得既正交于$\boldsymbol{v}_\perp$又正交于$\boldsymbol{n}$的第二个参考向量（根据左手拇指法则）。基于图3.3所示的三角关系和第1章练习14所得到的结论可知：

$$||\boldsymbol{n}\times\boldsymbol{v}|| = ||\boldsymbol{n}||\,||\boldsymbol{v}||\text{sin}\alpha = ||\boldsymbol{v}||\sin\alpha = ||\boldsymbol{v}_\perp||$$

其中，α是$\boldsymbol{n}$与$\boldsymbol{v}$之间的夹角。由此可知：两个参考向量的长度相等，且都位于旋转的圆周之上。根据三角学知识，我们就可以将这两个参考向量建立如下关系：

$$R_n(\boldsymbol{v}_\perp) = \cos\boldsymbol{\theta}\boldsymbol{v}_\perp + \sin\boldsymbol{\theta}(\boldsymbol{n}\times\boldsymbol{v})$$

这样就推导出了下列旋转公式：

$$\begin{aligned}
&R_n(\boldsymbol{v}) = \mathrm{proj}_n(\boldsymbol{v}) + R_n(\boldsymbol{v}_\perp)\\
&= (\boldsymbol{n}\cdot\boldsymbol{v})\boldsymbol{n} + \cos\boldsymbol{\theta}\boldsymbol{v}_\perp + \sin\boldsymbol{\theta}(\boldsymbol{n}\times\boldsymbol{v})\\
&= (\boldsymbol{n}\cdot\boldsymbol{v})\boldsymbol{n} + \cos\boldsymbol{\theta}(\boldsymbol{v} - (\boldsymbol{n}\cdot\boldsymbol{v})\boldsymbol{n}) + \sin\boldsymbol{\theta}(\boldsymbol{n}\times\boldsymbol{v})\\
&= \cos\boldsymbol{\theta}\boldsymbol{v} + 1 - \cos\boldsymbol{\theta})(\boldsymbol{n}\cdot\boldsymbol{v})\boldsymbol{n} + \sin\boldsymbol{\theta}(\boldsymbol{n}\times\boldsymbol{v})
\end{aligned} \qquad （3.5）$$

我们把证明此公式是一种线性变换的任务留作章末的习题。若要得

到旋转的变换矩阵表示，仅需按式（3.3）那样将各个标准基向量代入到$R_{\boldsymbol{n}}$中，再把得到的向量分别作为矩阵的行向量（见式（3.4））。最终得到的结果为：

$$\boldsymbol{R}_n = \begin{bmatrix} c+(1-c)x^2 & (1-c)xy+sz & (1-c)xz-sy \\ (1-c)xy-sz & c+(1-c)y^2 & (1-c)yz+sx \\ (1-c)xz+sy & (1-c)yz-sx & c+(1-c)z^2 \end{bmatrix}$$

此处设$c=\cos\boldsymbol{\theta}$且$s=\sin\boldsymbol{\theta}$。

旋转矩阵有个有趣的性质：每个行向量都为单位长度且两两正交（请分别证明）。也就是说，这些行向量都是规范正交的（orthonormal，即互相正交且具有单位长度）。若一个矩阵的行向量都是规范正交的，则称此矩阵为正交矩阵（orthogonal matrix）。正交矩阵有个引人注目的性质，即它的逆矩阵与转置矩阵是相等的：

$$\boldsymbol{R}_n^{-1} = \boldsymbol{R}_n^T = \begin{bmatrix} c+(1-c)x^2 & (1-c)xy-sz & (1-c)xz+sy \\ (1-c)xy+sz & c+(1-c)y^2 & (1-c)yz-sx \\ (1-c)xz-sy & (1-c)yz+sx & c+(1-c)z^2 \end{bmatrix}$$

通常来说，由于正交矩阵的逆矩阵计算方便且高效，所以很受青睐。

特别地，如果选择绕x轴、y轴或z轴进行旋转（即分别取$\boldsymbol{n}=(1,0,0)$、$\boldsymbol{n}=(0,1,0)$和$\boldsymbol{n}=(0,0,1)$），便会获得以x、y、z为旋转轴的对应旋转矩阵：

$$\boldsymbol{R}_x=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\boldsymbol{\theta} & \sin\boldsymbol{\theta} & 0\\ 0 & -\sin\boldsymbol{\theta} & \cos\boldsymbol{\theta} & 0\\ 0 & 0 & 0 & 1\end{bmatrix},\ \boldsymbol{R}_y=\begin{bmatrix}\cos\boldsymbol{\theta} & 0 & -\sin\boldsymbol{\theta} & 0\\ 0 & 1 & 0 & 0\\ \sin\boldsymbol{\theta} & 0 & \cos\boldsymbol{\theta} & 0\\ 0 & 0 & 0 & 1\end{bmatrix},\ \boldsymbol{R}_z=\begin{bmatrix}\cos\boldsymbol{\theta} & \sin\boldsymbol{\theta} & 0 & 0\\ -\sin\boldsymbol{\theta} & \cos\boldsymbol{\theta} & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

例3.3

假设定义了一个最小点坐标为(−1, 0, −1)和最大点坐标为(1, 0, 1)的正方形。现在，我们希望令它绕y轴顺时针旋转−30°（即按逆时针方向旋转30°）。根据问题所述可知，$\boldsymbol{n}=(0,\ 1,\ 0)$，代入$\boldsymbol{R}_{\boldsymbol{n}}$并进行化简，可得到$y$轴的旋转矩阵为：

$$\boldsymbol{R}_y=\begin{bmatrix}\cos\boldsymbol{\theta} & 0 & -\sin\boldsymbol{\theta}\\ 0 & 1 & 0\\ \sin\boldsymbol{\theta} & 0 & \cos\boldsymbol{\theta}\end{bmatrix}=\begin{bmatrix}\cos(-30°) & 0 & -\sin(-30°)\\ 0 & 1 & 0\\ \sin(-30°) & 0 & \cos(-30°)\end{bmatrix}=\begin{bmatrix}\frac{\sqrt{3}}{2} & 0 & \frac{1}{2}\\ 0 & 1 & 0\\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2}\end{bmatrix}$$

为了旋转该正方形，还需将其最小点、最大点坐标分别乘以得到的旋转矩阵：

$$[-1,\ 0,\ -1]\begin{bmatrix}\frac{\sqrt{3}}{2} & 0 & \frac{1}{2}\\ 0 & 1 & 0\\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2}\end{bmatrix}\approx[-0.36,\ 0,\ -1.36]\quad[1,\ 0,\ 1]\begin{bmatrix}\frac{\sqrt{3}}{2} & 0 & \frac{1}{2}\\ 0 & 1 & 0\\ -\frac{1}{2} & 0 & \frac{\sqrt{3}}{2}\end{bmatrix}\approx[0.36,\ 0,\ 1.36]$$

结果如图3.4所示。

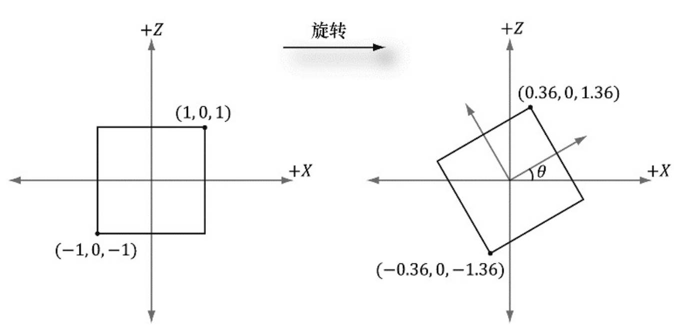


图3.4　令例3.3中的正方形绕y轴旋转$-30°$后的效果。注意，当沿着y轴正方向俯瞰的时候，由于$y=0$，因此示意图看上去是2D效果

3.2　仿射变换

3.2.1　齐次坐标

在下一节中我们将会看到，仿射变换（affine transformation）是由一个线性变换与一个平移变换组合而成的。对于向量而言，平移操作是没有意义的，因为向量只描述方向与大小、却与位置无关。换句话说，平移操作不应作用于向量。因此，平移变换只能应用于点（即位置向量）。齐次坐标（homogeneous coordinate）所提供的表示机制，使我们可以方便地对点和向量进行统一的处理。在采用齐次坐标表示法时，我们将坐标扩充为四元组，其中，第四个坐标w的取值将根据被描述对象是点还是向量而定。具体来讲：

1．$(x, y, z, 0)$表示向量；

2．$(x, y, z, 1)$表示点。

在后面我们将会证明：设$w = 1$能使点被正确地平移，设$w = 0$则可以防止向量坐标受到平移操作的影响（我们不希望对向量的坐标进行平移变换，因为这个计算过程会改变它的方向和大小——而平移操作不应当修改向量的任何一种“属性”）。

注意　Note

齐次坐标表示法与图1.17所示的思路一致，即两点之差 $\boldsymbol{q}-\boldsymbol{p}=(q_x,q_y,q_z,1)-(p_x,p_y,p_z,1)=(q_x-p_x,q_y-p_y,q_z-p_z,0)$得到的是一个向量，而一个点与一个向量之和 $\boldsymbol{p}+\boldsymbol{v}=(p_x,p_y,p_z,1)+(v_x,v_y,v_z,0)=(p_x+v_x,p_y+v_y,p_z+v_z,1)$得到的是一个点。

3.2.2 仿射变换的定义及其矩阵表示

线性变换并不能表示出我们需要的所有变换，因此，现将其扩充为一种称作仿射变换的映射范围更广的函数类。仿射变换为一个线性变换加上一个平移向量$\boldsymbol{b}$，即

$$\alpha(\boldsymbol{u})=\tau(\boldsymbol{u})+\boldsymbol{b}$$

或者用矩阵表示法

$$\alpha(\boldsymbol{u})=\boldsymbol{u}\boldsymbol{A}+\boldsymbol{b}=[x,\ y,\ z]\begin{bmatrix}A_{11} & A_{12} & A_{13}\\ A_{21} & A_{22} & A_{23}\\ A_{31} & A_{32} & A_{33}\end{bmatrix}+[b_x,\ b_y,\ b_z]=[x',\ y',\ z']$$

其中，$\boldsymbol{A}$是一个线性变换的矩阵表示。

如果用$w=1$把坐标扩充为齐次坐标，那么就可以将上式更简洁地写作：

$$[x,\ y,\ z,\ 1]\begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 \\ A_{21} & A_{22} & A_{23} & 0 \\ A_{31} & A_{32} & A_{33} & 0 \\ b_x & b_y & b_z & 1 \end{bmatrix} = [x',\ y',\ z',\ 1] \quad (3.6)$$

式（3.6）中的4×4矩阵称为仿射变换的矩阵表示。

可以看出，加上向量$\boldsymbol{b}$的这步运算，从本质上来说是一种平移操作（使目标对象的位置发生了改变）。但是，我们既不希望将此平移操作应用到向量上（因为向量的性质中并没有位置这个概念），又想令向量受到仿射变换中线性部分的处理。此时，如果将向量的第四个分量设为$w = 0$，它便不会受到向量$\boldsymbol{b}$平移操作的影响（可通过矩阵的乘法运算来验证这一点）。

注意 Note

由于行向量与上述4×4仿射变换矩阵的第四列点积有$[x, y, z, w] \cdot [0, 0, 0, 1] = w$，所以此矩阵不会改变输入向量的$w$坐标。

3.2.3　平移

恒等变换（identity transformation）是一种直接返回其输入参数的线性变换，形如$\boldsymbol{I}(\boldsymbol{u}) = \boldsymbol{u}$。不难证明，这种线性变换的矩阵表示即为单

位矩阵。

现将平移变换（translation transformation）定义为仿射变换，此时，其中的线性变换就是一种恒等变换，即

$$\tau(\boldsymbol{u})=\boldsymbol{u}\boldsymbol{I}+\boldsymbol{b}=\boldsymbol{u}+\boldsymbol{b}$$

如您所见，此线性变换简单地利用向量$\boldsymbol{b}$对点$\boldsymbol{u}$进行平移（或位移）。图3.5详细地展示了物体位移的过程——物体上的每个点都通过同一向量$\boldsymbol{b}$进行了平移。

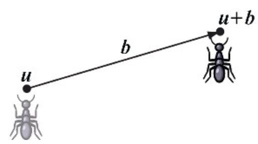


图3.5　借助位移向量$\boldsymbol{b}$ 对蚂蚁进行位移

根据式（3.6）可知，τ的矩阵表示为：

$$\boldsymbol{T}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\b_x&b_y&b_z&1\end{bmatrix}$$

该矩阵称为平移矩阵（translation matrix）。

平移矩阵的逆矩阵则为：

$$\boldsymbol{T}^{-1}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\-b_x&-b_y&-b_z&1\end{bmatrix}$$

例3.4

假设定义了一个正方形，其最小点坐标为(−8, 2, 0)，最大点坐标为(−2, 8, 0)。我们希望将此正方形沿x轴正方向平移12个单位，沿y轴正方向平移−10个单位，在z轴方向保持不变，则其对应的平移矩阵为：

$$\boldsymbol{T}=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 12 & -10 & 0 & 1\end{bmatrix}$$

现对此正方形进行平移（变换），将其最小点、最大点坐标分别乘以上述平移矩阵：

$$[-8,\ 2,\ 0,\ 1]\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 12 & -10 & 0 & 1\end{bmatrix}=[4,\ \ -8,\ \ 0,\ \ 1]$$

$$[-2,\ 8,\ 0,\ 1]\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 12 & -10 & 0 & 1\end{bmatrix}=[10,\ \ -2,\ \ 0,\ \ 1]$$

平移结果如图3.6所示。

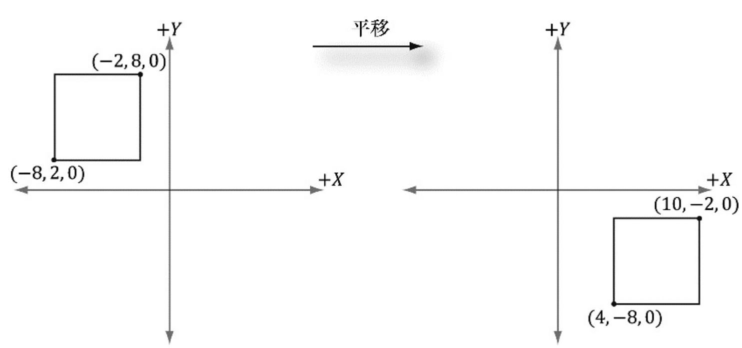


图3.6　将例3.4中的正方形在x轴上平移12个单位长度，在y轴上平移−10个单位长度。注意，在向z轴负方向俯瞰的时候，由于$z = 0$，因此几何图示呈2D效果

注意　Note

设$\boldsymbol{T}$为一个变换矩阵，我们可通过计算乘积$\boldsymbol{vT} = \boldsymbol{v}'$来变换一个点或向量。不难看出，若通过$\boldsymbol{T}$对点或向量进行变换之后，再经其逆矩阵$\boldsymbol{T}^{-1}$的变换，那么，最终会得到初始的向量：$\boldsymbol{vTT}^{-1} = \boldsymbol{vI} = \boldsymbol{v}$。换句话说，逆变换能够“还原”变换操作。例如，如果将一个点在x轴正方向平移5个单位，再通过逆变换，令该点沿x轴正方向平移−5个单位，那么，此点最终会回到初始的位置。又如，若令一个点绕y轴旋转30°，再根据逆变换使该点绕y轴旋转−30°，则此点最终也将回到起始位置。总的来说，变换矩阵及其逆矩阵的作用正相反。这两种变换的复合，从几何学的角度上来看使物体保持不变。

3.2.4　缩放和旋转的仿射矩阵

通过观察可以发现，如果$\boldsymbol{b} = 0$，则仿射变换将退化为线性变换。这样一来，我们就能用$\boldsymbol{b} = 0$的仿射变换来表示任意线性变换。更进一步说，也就意味着仅通过一个4×4的仿射矩阵表达出任意的线性变换。例如，缩放矩阵与旋转矩阵可写作下列的4×4矩阵：

$$\boldsymbol{S}=\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{R}_n=\begin{bmatrix} c+(1-c)x^2 & (1-c)xy+sz & (1-c)xz-sy & 0 \\ (1-c)xy-sz & c+(1-c)y^2 & (1-c)yz+sx & 0 \\ (1-c)xz+sy & (1-c)yz-sx & c+(1-c)z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

如此一来，就能用4×4矩阵统一地表示所有变换，并通过1×4齐次行向量来表示点和向量。

3.2.5 仿射变换矩阵的几何意义

经过本节的学习，我们对仿射变换矩阵中各项数据几何意义上的直觉将得到进一步的提升。首先让我们来考察刚体变换（rigid body transformation），其本质是一种保形（shape preserving，即保持形状）变换。以下便是刚体变换在现实生活中的一个例子：从书桌上拿起书，再将它放到书架上。在移动书的这个过程中（平移），很可能会改变它的朝向（旋转）。设τ为描述物体旋转操作的旋转变换，而$\boldsymbol{b}$为定义物体平移操作的平移向量。那么，刚体变换就可以用仿射变换来表示：

$$\alpha(x,y,z)=\tau(x,y,z)+\boldsymbol{b}=x\tau(\boldsymbol{i})+y\tau(\boldsymbol{j})+z\tau(\boldsymbol{k})+\boldsymbol{b}$$

在矩阵表示法中，若采用齐次坐标（表示点时，$w=1$；表示向量时，$w=0$，如此一来，平移变换就不会作用于向量），上式将被改写为：

$$[\ x,\ y,\ z,\ \mathrm{w}\ ]\begin{bmatrix}\leftarrow \tau(\boldsymbol{i}) \rightarrow \\ \leftarrow \tau(\boldsymbol{j}) \rightarrow \\ \leftarrow \tau(\boldsymbol{k}) \rightarrow \\ \leftarrow \boldsymbol{b} \rightarrow\end{bmatrix} = [\ x',\ y',\ z',w\ ] \tag{3.7}$$

至此，为了理解此方程的几何意义，我们还要将矩阵中的行向量依次绘制出来（见图3.7）。由于τ是一个旋转变换，所以它具有保长性与保角性（详见章末习题26）。特别是我们能看到τ仅将标准基向量$\boldsymbol{i}$、$\boldsymbol{j}$和$\boldsymbol{k}$，分别旋转到对应的新方向$\tau(\boldsymbol{i})$、$\tau(\boldsymbol{j})$和$\tau(\boldsymbol{k})$。而向量$\boldsymbol{b}$则是一个位置向量，它表示物体相对于原点的位移。现在来看图3.7，它以几何学的角度展示了如何通过计算$\alpha(x,\ y,\ z) = x\tau(\boldsymbol{i}) + y\tau(\boldsymbol{j}) + z\tau(\boldsymbol{k}) + \boldsymbol{b}$来求取变换后的点。

这种思路同样可以运用在缩放或斜切（skew，也有译作倾斜、扭曲等）变换上。请考虑这样一种线性变换τ，它将图3.8所示的正方形拉扯为一个平行四边形。斜切处理后的点即为斜切变换后的基向量的线性组合。

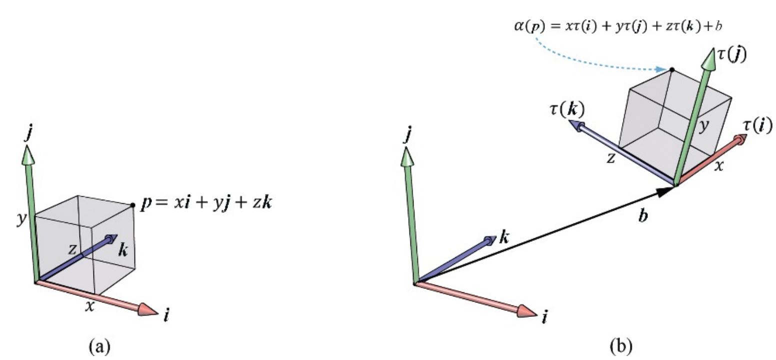


图3.7　仿射变换矩阵中行向量的几何意义。点$\alpha(\boldsymbol{p})$是变换后的基向量$\tau(\boldsymbol{i})$、$\tau(\boldsymbol{j})$、$\tau(\boldsymbol{k})$与偏移向量$\boldsymbol{b}$的线性组合

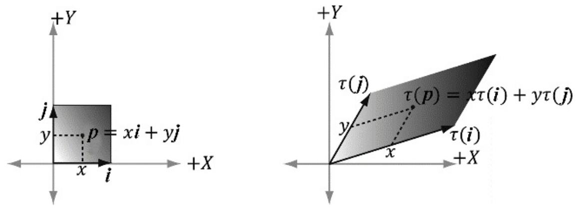


图3.8　对于一种把正方形拉扯为平行四边形的线性变换而言，经过变换的点$\tau(\boldsymbol{p}) = (x,y)$即为斜切变换后的基向量$\tau(\boldsymbol{i})$与$\tau(\boldsymbol{j})$的线性组合

3.3 变换的复合[2]

假设$\boldsymbol{S}$是一个缩放矩阵，$\boldsymbol{R}$是一个旋转矩阵，$\boldsymbol{T}$是一个平移矩阵。现给定一个由8个顶点$\boldsymbol{v}_i$（其中$i = 0, 1, \cdots, 7$）构成的立方体，并希望将这3种变换相继应用到此正方体的每个顶点之上。我们以下列简明的方式来逐步对顶点进行变换：

$$((\boldsymbol{v}_i\boldsymbol{S})\boldsymbol{R})\boldsymbol{T} = (\boldsymbol{v}_i'\boldsymbol{R})\boldsymbol{T} = \boldsymbol{v}_i''\boldsymbol{T} = \boldsymbol{v}_i''' \text{ 其中 } i = 0, 1, \cdots, 7$$

然而，由于矩阵乘法满足结合律，因而此式可以等价地改写为：

$$\boldsymbol{v}_i(\boldsymbol{SRT}) = \boldsymbol{v}_i''' \text{ 其中 } i = 0, 1, \cdots, 7$$

还可将$\boldsymbol{C} = \boldsymbol{SRT}$视为一个矩阵，即提前将3种变换封装为一个净变换矩阵。换句话说，矩阵之间的乘法法则使我们得以将不同的变换连接在一起。

这里实际还涉及性能问题。来看一个例子：假设有一个由20000个点组成的3D物体，我们希望将上述3种几何变换，逐个作用到这个物体上。如果采用按部就班的计算方法，我们需要进行20000×3次向量与矩阵的乘法运算。但通过上述组合矩阵的计算方法，只需要执行20000次的向量与矩阵乘法运算以及两次矩阵与矩阵的乘法运算即可。显而易见的是，比起前者中近3倍的大量向量与矩阵乘法运算而言，后者中两次额外的矩阵与矩阵乘法运算真可谓是九牛一毛。

注意 Note

这里要再次指出：矩阵之间的乘法运算不满足交换律。这一点亦可以从几何角度上看出。例如，一个旋转操作后面跟有一个平移变换，我们能够用矩阵的乘积$\boldsymbol{RT}$来表示。而采用同样的变换矩阵，先平移后旋转，即$\boldsymbol{TR}$，它的变换结果却与$\boldsymbol{RT}$全然不同。图3.9演示了这两种变换过程的差异。

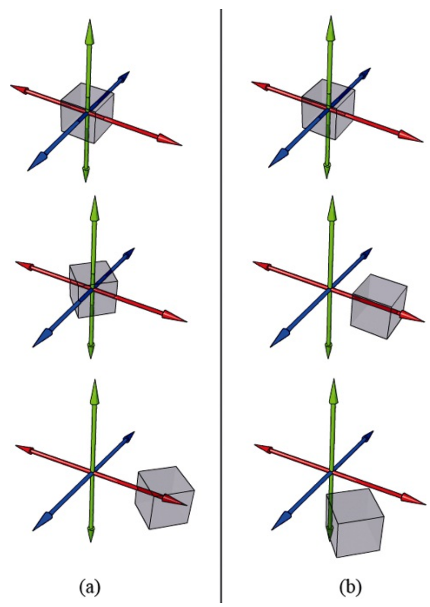


图3.9　两种变换过程的差异比较

（a）先旋转后平移　（b）先平移后旋转

3.4 坐标变换[3]

众所周知，100℃表示的是摄氏温标下水的沸点。那么，怎样用华氏温标来描述这同样环境下水的沸点呢？换言之，若用华氏温标表示水的沸点，其对应的标量值是多少呢？为了实现此转换（或称标架的变换），我们需要知道摄氏度和华氏度这两种温标之间的关系。这两者之间的转换公式是$T_\mathrm{F}=\frac{9}{5}T_\mathrm{C}+32^\circ$。由此可知，水的沸点换算为华氏度为$T_\mathrm{F}=\frac{9}{5}(100)^\circ+32^\circ=212^\circ$。

通过这个例子可知：根据标架A与另一不同标架B的关系，我们就可以将相对于标架A表示某量的标量k，转换为相对于标架B描述同一种量的新标量k'。在后续子小节中，我们会遇到类似的问题，但届时我们研究的并不再是标量，而是要相对于不同的标架来转换点或向量的坐标（见图3.10）。我们把不同标架间的坐标的转换称之为坐标变换（change of coordinate transformation）。

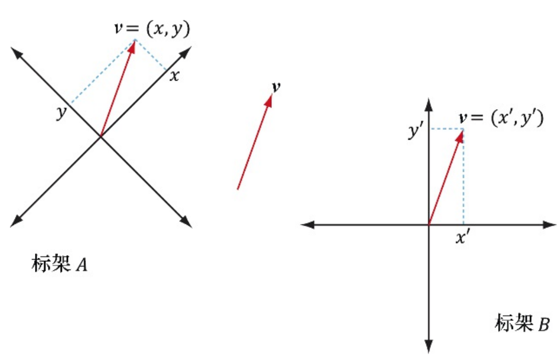


图3.10　同一个向量$\boldsymbol{v}$在不同的标架中有着不同的坐标。此向量在标架A中的坐标为(x, y)，在标架B中的坐标则为(x', y')

值得注意的是，在坐标变换的过程中，几何体本身并没有随之发生改变。坐标变换改变的仅是物体的参考系（又称参照系），因此改变的实为几何体的坐标表示。相比之下，我们可以认为旋转、平移和缩放这些操作才使几何体发生了实质上的移动或形变。

在3D计算机图形学中，我们往往会用到许多不同的坐标系，所以需要了解在它们之间互相转换坐标的方法。由于位置是点的属性，与向量无关，所以点和向量的坐标变换是不同的，下面就此展开开讨论。

3.4.1　向量的坐标变换

思考图3.11，其中有一向量$\boldsymbol{p}$分别位于标架A和标架B之中。假设给定向量$\boldsymbol{p}$在标架A中的坐标为$\boldsymbol{p}_A=(x,y)$，现希望求得向量$\boldsymbol{p}$在标架B中的对应坐标$\boldsymbol{p}_B=(x',y')$。换言之，如果给出了一个向量在某一个标架中的坐标，那么如何求出该向量在另一个不同标架中的对应坐标呢？

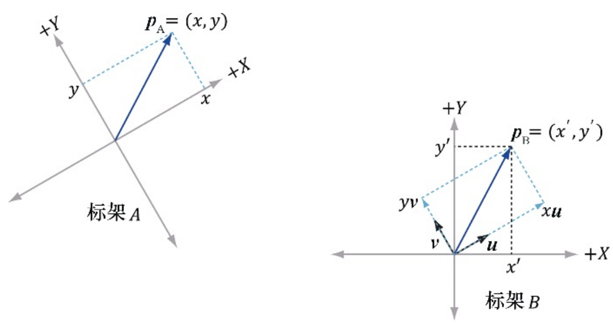


图3.11　求取向量$\boldsymbol{p}$在标架B中坐标的几何示意图

从图3.11中可知：

$$\boldsymbol{p}=x\boldsymbol{u}+y\boldsymbol{v}$$

其中，$\boldsymbol{u}$和$\boldsymbol{v}$分别是标架中x轴、y轴正方向上的单位向量。用标架B中的坐标来表示以上公式中的单位向量，可得：

$$\boldsymbol{p}_B = x\boldsymbol{u}_B + y\boldsymbol{v}_B$$

所以，如果给定$\boldsymbol{p}_A = (x, y)$，也已知向量$\boldsymbol{u}$和向量$\boldsymbol{v}$相对于标架$B$的坐标分别为$\boldsymbol{u}_B = (u_x, u_y)$以及$\boldsymbol{v}_B = (v_x, v_y)$，那么就一定能求出$\boldsymbol{p}_B = (x', y')$。

现将向量的坐标变换推广到3D空间，如果$\boldsymbol{p}_A = (x, y, z)$，那么

$$\boldsymbol{p}_B = x\boldsymbol{u}_B + y\boldsymbol{v}_B + z\boldsymbol{w}_B$$

其中，$\boldsymbol{u}$、$\boldsymbol{v}$和$\boldsymbol{w}$分别是指向标架中x轴、y轴和z轴正方向上的单位向量。

3.4.2 点的坐标变换

点与向量的坐标变换稍有不同，这是由于位置是点的一个重要属性，因而不能将图3.11所示的向量平移方法应用于点上。

图3.12展示了一个对点进行坐标变换的情景，通过观察，可以将点$\boldsymbol{p}$表示为：

$$\boldsymbol{p} = x\boldsymbol{u} + y\boldsymbol{v} + \boldsymbol{Q}$$

其中$\boldsymbol{u}$和$\boldsymbol{v}$是分别指向标架A中x轴和y轴正方向上的单位向量，且$\boldsymbol{Q}$是标架A中的原点。用标架B中的坐标来表示上式中的每一个向量和点，可得：

$$\boldsymbol{p}_B = x\boldsymbol{u}_B + y\boldsymbol{v}_B + \boldsymbol{Q}_B$$

如果给出$\boldsymbol{p}_A=(x,y)$，同时也知道向量$\boldsymbol{u}$、$\boldsymbol{v}$以及原点$\boldsymbol{Q}$相对于标架B的坐标分别为$\boldsymbol{u}_B=(u_x,u_y)$、$\boldsymbol{v}_B=(v_x,v_y)$以及$\boldsymbol{Q}_B=(Q_x,Q_y)$，那么，我们总能求出$\boldsymbol{p}_B=(x',y')$。

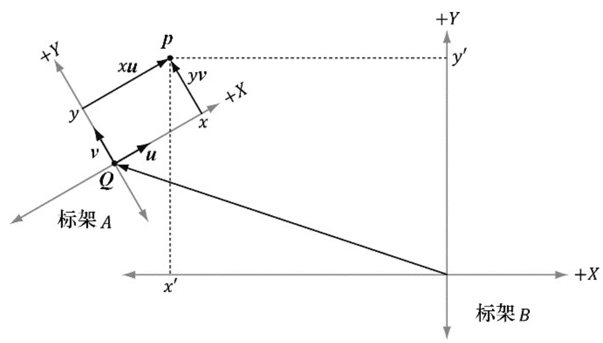


图3.12　求取点$\boldsymbol{p}$在标架B中坐标的几何示意图

现把点的坐标变换推广到3D空间，如果$\boldsymbol{p}_A=(x,y,z)$，那么

$$\boldsymbol{p}_B=x\boldsymbol{u}_B+y\boldsymbol{v}_B+z\boldsymbol{w}_B+\boldsymbol{Q}_B$$

其中，向量$\boldsymbol{u}$、$\boldsymbol{v}$和$\boldsymbol{w}$是分别指向标架A中x轴、y轴和z轴的正方向上的单位向量，$\boldsymbol{Q}$则为标架A中的原点。

3.4.3　坐标变换的矩阵表示

到目前为止，我们已经分别探讨了向量和点的坐标变换：

$(x',y',z')=x\boldsymbol{u}_B+y\boldsymbol{v}_B+z\boldsymbol{w}_B$　　对于向量而言

$(x',y',z')=x\boldsymbol{u}_B+y\boldsymbol{v}_B+z\boldsymbol{w}_B+\boldsymbol{Q}_B$　　对于点而言

如果使用齐次坐标，就可以用同一公式对点和向量进行处理：

$$(x', y', z', w) = x\boldsymbol{u}_B + y\boldsymbol{v}_B + z\boldsymbol{w}_B + \boldsymbol{Q}_B \qquad (3.8)$$

如果$w=0$，此式化简为向量的坐标变换公式；如果$w=1$，则此式化简为点的坐标变换公式。式（3.8）的优点在于：只要为其正确地设置w分量值，它就能相应地处理点或向量的坐标变换。这样一来，我们也就无须分别记住两个公式了（一个处理向量，另一个处理点）。根据式（2.3），可以将式（3.8）改写为矩阵形式：

$$\begin{aligned}
[x', y', z', w] &= [x, y, z, w]\begin{bmatrix} \leftarrow \boldsymbol{u}_B \rightarrow \\ \leftarrow \boldsymbol{v}_B \rightarrow \\ \leftarrow \boldsymbol{w}_B \rightarrow \\ \leftarrow \boldsymbol{Q}_B \rightarrow \end{bmatrix} \\
&= [x, y, z, w]\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & Q_z & 1 \end{bmatrix} \\
&= x\boldsymbol{u}_B + y\boldsymbol{v}_B + z\boldsymbol{w}_B + w\boldsymbol{Q}_B \qquad (3.9)
\end{aligned}$$

其中，$\boldsymbol{Q}_B = (Q_x, Q_y, Q_z, 1)$$\boldsymbol{u}_B = (u_x, u_y, u_z, 0)$$\boldsymbol{v}_B = (v_x, v_y, v_z, 0)$与$\boldsymbol{w}_B = (w_x, w_y, w_z, 0)$分别表示标架$A$中的原点和诸坐标轴相对于标架$B$的齐次坐标。我们把式（3.9）里能把标架$A$中的坐标转换（或映射，map）为标架$B$中坐标的$4\times 4$矩阵，称为坐标变换矩阵（change of coordinate matrix）或标架变换矩阵（change of frame matrix）。

3.4.4　坐标变换矩阵及其结合律

现假设有3个标架F、G和H。$\boldsymbol{A}$为将坐标从F转换到G的标架变换矩阵，$\boldsymbol{B}$为把坐标由G转换至H的标架变换矩阵。在标架F中，有一向量的

坐标为$\boldsymbol{p}_F$，我们希望求出此向量相对于标架H的坐标$\boldsymbol{p}_H$。可按一般顺序来逐步计算：

$$(\boldsymbol{p}_F\boldsymbol{A})\boldsymbol{B}=\boldsymbol{p}_H$$

$$(\boldsymbol{p}_G)\boldsymbol{B}=\boldsymbol{p}_H$$

由于矩阵的乘法运算满足结合律，我们可将$(\boldsymbol{p}_F\boldsymbol{A})\boldsymbol{B}=\boldsymbol{p}_H$写作：

$$\boldsymbol{p}_F(\boldsymbol{AB})=\boldsymbol{p}_H$$

这样一来，就能把矩阵乘积$\boldsymbol{C}=\boldsymbol{AB}$看作是将坐标从标架$F$直接变换至标架$H$的标架变换矩阵，它将变换矩阵$\boldsymbol{A}$和$\boldsymbol{B}$结合为一个净矩阵——其思路就类似于函数的复合。

这种计算方法还会对性能产生影响。为了说明这一点，我们假设一个由20000个点构成的3D物体，现要对它的每个点进行两次标架变换。若使用逐步计算的方法，我们需进行20000×2次向量与矩阵的乘法运算。但利用结合矩阵的方法，我们只要进行20000次向量与矩阵的乘法运算以及一次矩阵与矩阵的乘法运算（用于结合两个标架变换矩阵）即可。不难看出，仅借助一次开销极低的矩阵之间的乘法运算，便可以节省多次向量与矩阵乘法所需的大量计算资源。

注意 Note

重要的事情说3遍：矩阵乘法是不满足交换律的，所以请不要认为矩阵乘积$\boldsymbol{AB}$与矩阵乘积$\boldsymbol{BA}$表示的是相同的复合变换。具体来说，矩

阵相乘的顺序就是变换的顺序。一般而言，它们的处理顺序是不能随意调换的。

3.4.5　坐标变换矩阵及其逆矩阵

假设给定向量$\boldsymbol{p}$相对于标架B的坐标$\boldsymbol{p}_B$，以及将坐标由标架A转换到标架B的变换矩阵$\boldsymbol{M}$，即有$\boldsymbol{p}_B = \boldsymbol{p}_A\boldsymbol{M}$。现希望求得$\boldsymbol{p}_A$。换句话说，我们这次是希望通过坐标变换矩阵，将标架B中的坐标映射到标架A中。为了求出这个矩阵，我们假设矩阵$\boldsymbol{M}$是可逆的（即存在$\boldsymbol{M}^{-1}$）。通过下列步骤，我们就能得到坐标$\boldsymbol{p}_A$：

$$\boldsymbol{p}_B = \boldsymbol{p}_A\boldsymbol{M}$$

$$\boldsymbol{p}_B\boldsymbol{M}^{-1} = \boldsymbol{p}_A\boldsymbol{M}\boldsymbol{M}^{-1}$$　等式左右两侧同时乘以矩阵$\boldsymbol{M}^{-1}$

$$\boldsymbol{p}_B\boldsymbol{M}^{-1} = \boldsymbol{p}_A\boldsymbol{I}$$　由逆矩阵的定义可知$\boldsymbol{M}\boldsymbol{M}^{-1} = \boldsymbol{I}$

$$\boldsymbol{p}_B\boldsymbol{M}^{-1} = \boldsymbol{p}_A$$　由单位矩阵的定义可知$\boldsymbol{p}_A\boldsymbol{I} = \boldsymbol{p}_A$

所以矩阵$\boldsymbol{M}^{-1}$即为将坐标由标架B转换到标架A的变换矩阵。

图3.13说明了坐标变换矩阵及其逆矩阵之间的关系。还有一点要注意的是，本书内所有的标架变换映射都是可逆的，因此我们不必担心逆矩阵是否存在这一问题。

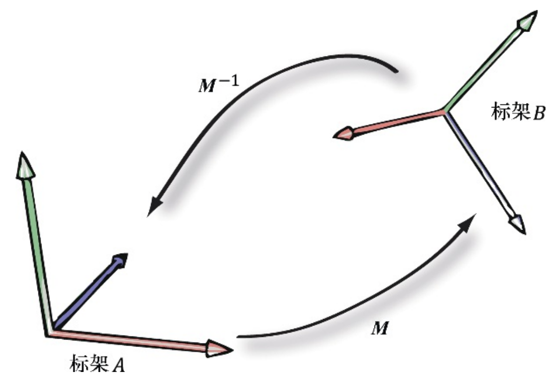


图3.13　矩阵$\boldsymbol{M}$把标架A中的坐标映射到标架B，矩阵$\boldsymbol{M}^{-1}$则把标架B中的坐标映射到标架A

图3.14展示了如何用坐标变换矩阵来解释逆矩阵的性质($\boldsymbol{AB})^{-1}=\boldsymbol{B}^{-1}\boldsymbol{A}^{-1}$。

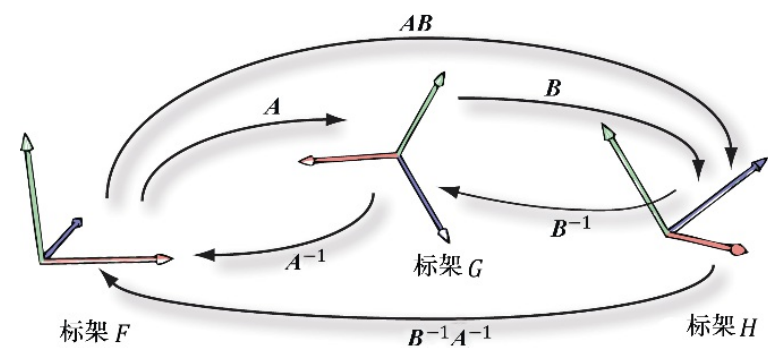


图3.14　矩阵$\boldsymbol{A}$把坐标从标架F映射到标架G，矩阵$\boldsymbol{B}$把坐标从标架G映射到标架H，矩阵乘积

$\boldsymbol{AB}$则把坐标从标架F直接映射至标架H。矩阵$\boldsymbol{B}^{-1}$把坐标从标架H映射到标架G，矩阵$\boldsymbol{A}^{-1}$把坐标从标架G映射到标架F，利用矩阵乘积$\boldsymbol{B}^{-1}\boldsymbol{A}^{-1}$则可把坐标直接从标架$H$映射至标架$F$

3.5　变换矩阵与坐标变换矩阵

到目前为止，我们已经对“使几何体本身发生改变”的变换（缩放、旋转和平移）与坐标变换进行了区分。在本节中，我们将证明：从数学角度上看，两者在数学上其实是等价的。即，可将改变几何体的变换解释为坐标变换，反之亦然。

图3.15展示了式（3.7）中的行向量（令物体先旋转再平移的仿射变换矩阵）与式（3.9）中的行向量（坐标变换矩阵）在几何意义上的相似性。

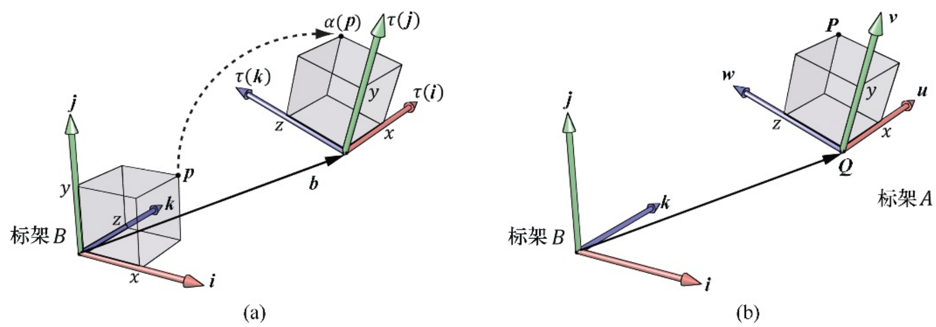


图3.15　从图中可以对比看出$\boldsymbol{b}=\boldsymbol{Q}$，$\tau(\boldsymbol{i})=\boldsymbol{u}$，$\tau(\boldsymbol{j})=\boldsymbol{v}$，$\tau(\boldsymbol{k})=\boldsymbol{w}$。图a中，假设我们在当前的工作中所采用的坐标系为标架B，现在要运用仿射变换相对于标架B来改变图内立方体的位置以及朝向：$\alpha(x,y,z,w)=x\tau(\boldsymbol{i})+y\tau(\boldsymbol{j})+z\tau(\boldsymbol{k})+w\boldsymbol{b}$。图b中，我们有标架$A$与标架$B$两个坐标系。通过公式$\boldsymbol{p}_B=x\boldsymbol{u}_B+y\boldsymbol{v}_B+z\boldsymbol{w}_B+w\boldsymbol{Q}_B$就可以把相对于标架$A$组成立方体的诸点坐标转换为标架$B$中的坐标，其中$\boldsymbol{p}_A=(x,y,z,w)$。在这两种情形之中，我们可以分别得到相对于标架$B$中的坐标$\alpha(\boldsymbol{p})=(x',y',z',w)=\boldsymbol{p}_B$

如果我们能认识到这一点，会发现其意义非凡。对于坐标变换来说，标架之间的差异仅为位置和朝向。因此，令坐标在标架之间转换的数学公式实质上描述的即为所需执行的旋转和平移操作，最终也会得到

与几何变换（对物体进行缩放、旋转和平移）相同的数学形式，可谓殊途同归。几何变换也好，坐标变换也罢，计算出的结果都是相同的，差别只在于解释变换的角度。对于某些情况来说，保持物体不变，使之在多个坐标系之间转换是种更直观的办法。但是，若描述对象的参考系发生了变化，则物体的坐标表示也会随之改变（图3.15b演示了这种情景）。有些时候，我们又希望在同一个坐标系中表示物体的变换，而不改变其参考系（这种情况可参见图3.15a），此时即可采用几何变换法。

注意 Note

这段讨论证实了我们能够将一个改变几何体的复合变换（缩放、旋转和平移），解释为一种对应的坐标变换。由于我们以后通常要将世界空间（第5章）的坐标变换矩阵定义为缩放、旋转和平移操作组成的复合变换，所以了解这一点是很重要的。

3.6 DirectXMath库提供的变换函数

本节我们对DirectXMath库中与变换相关的函数进行总结，以供参考。

```
// 构建一个缩放矩阵：
XMMATRIX XM_CALLCONV XMMatrixScaling(
float ScaleX,
float ScaleY,
float ScaleZ);                          // 缩放系数

// 用一个3D向量中的分量来构建缩放矩阵：
XMMATRIX XM_CALLCONV XMMatrixScalingFromVector(
FXMVECTOR Scale);                       // 缩放系数(sx,sy, sz)

// 构建一个绕x轴旋转的矩阵Rx：
XMMATRIX XM_CALLCONV XMMatrixRotationX(
    float Angle);                       // 以顺时针方向按弧度θ进行旋转

// 构建一个绕y轴旋转的矩阵Ry：
XMMATRIX XM_CALLCONV XMMatrixRotationY(
    float Angle);                       // 以顺时针方向按弧度θ进行旋转

// 构建一个绕z轴旋转的矩阵Rz：
XMMATRIX XM_CALLCONV XMMatrixRotationZ(
    float Angle);                       // 以顺时针方向按弧度θ进行旋转

// 构建一个绕任意轴旋转的矩阵Rn：
XMMATRIX XM_CALLCONV XMMatrixRotationAxis(
 FXMVECTOR Axis,                        // 旋转轴n
 float Angle);                          // 沿n轴正方向看，以顺时针方向按弧度θ进
行旋转

// 构建一个平移矩阵：
XMMATRIX XM_CALLCONV XMMatrixTranslation(
float OffsetX,
float OffsetY,
float OffsetZ);                         // 平移系数

// 用一个3D向量中的分量来构建平移矩阵：
XMMATRIX XM_CALLCONV XMMatrixTranslationFromVector(
```

```
FXMVECTOR Offset);                  // 平移系数(tx, ty, tz)

// 计算向量与矩阵的乘积vM，此函数为针对点的变换，即总是默认令vw = 1:
XMVECTOR XM_CALLCONV XMVector3TransformCoord(
FXMVECTOR V,     // 输入向量v
CXMMATRIX M);    // 输入矩阵M

// 计算向量与矩阵的乘积vM，此函数为针对向量的变换，即总是默认令vw = 0:
XMVECTOR XM_CALLCONV XMVector3TransformNormal(
FXMVECTOR V,      // 输入向量v
CXMMATRIX M);     // 输入矩阵M
```

对于最后的两个函数来说，用户不必显式设置w分量，因为在执行XMVector3TransformCoord时，默认$v_w = 1$，而当执行XMVector3TransformNormal时，默认$v_w = 0$。

3.7　小结

1．缩放、平移和旋转这3种基础操作的变换矩阵分别为：

$$\boldsymbol{S}=\begin{bmatrix}s_x & 0 & 0 & 0\\ 0 & s_y & 0 & 0\\ 0 & 0 & s_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\boldsymbol{T}=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ b_x & b_y & b_z & 1\end{bmatrix}$$

$$\boldsymbol{R}_n=\begin{bmatrix}c+(1-c)x^2 & (1-c)xy+sz & (1-c)xz-sy & 0\\ (1-c)xy-sz & c+(1-c)y^2 & (1-c)yz+sx & 0\\ (1-c)xz+sy & (1-c)yz-sx & c+(1-c)z^2 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

2．我们通过4×4矩阵来表示变换，并利用1×4齐次坐标来描述点和向量：当把第4个分量设置为$w=1$时，表示点；设置为$w=0$时，则表示向量。这样一来，平移操作将只应用于点，而不会影响向量。

3．如果一个矩阵内所有的行向量都是单位长度且两两正交，则此矩阵为正交矩阵。正交矩阵有个特殊性质：它的逆矩阵与转置矩阵相等。因此，这使它的逆矩阵计算起来方便且高效。另外，所有的旋转矩阵皆为正交矩阵。

4．由于矩阵的乘法运算满足结合律，因此我们就能够将若干种变换矩阵合而为一。此矩阵给予物体的变换效果，与合成它的多个单一矩阵对物体按次序进行变换的净效果相同。

5．设$\boldsymbol{Q}_B$、$\boldsymbol{u}_B$、$\boldsymbol{v}_B$和$\boldsymbol{w}_B$分别表示标架A中的原点、x轴、y轴和z轴

相对于标架B的坐标。如果一个向量（或点）$\boldsymbol{p}$相对于标架A的坐标为$\boldsymbol{p}_A=(x,y,z)$，那么，此同一向量（或点）相对于标架B的坐标为：

（a）$\boldsymbol{p}_B=(x',y',z')=x\boldsymbol{u}_B+y\boldsymbol{v}_B+z\boldsymbol{w}_B$　　针对向量（具有大小和方向两种属性）而言

（b）$\boldsymbol{p}_B=(x',y',z')=\boldsymbol{Q}_B+x\boldsymbol{u}_B+y\boldsymbol{v}_B+z\boldsymbol{w}_B$　　针对位置向量（即点）而言

这些坐标变换还可以写为由齐次坐标组成的矩阵形式。

6．假设有3个标架F、G和H。已知将坐标由F转换到G的标架变换矩阵为$\boldsymbol{A}$，把坐标由G转换到H的标架变换矩阵为$\boldsymbol{B}$。根据矩阵与矩阵的乘法运算法则，可以将矩阵乘积$\boldsymbol{C}=\boldsymbol{AB}$看作把坐标由$F$直接转换到$H$的标架变换矩阵。这就是说，利用矩阵之间的乘法运算，能够将矩阵$\boldsymbol{A}$和矩阵$\boldsymbol{B}$的变换效果组合为一个净矩阵，并可记作$\boldsymbol{p}_F(\boldsymbol{AB})=\boldsymbol{p}_H$。

7．如果矩阵$\boldsymbol{M}$可以将坐标从标架A映射至标架B，那么，矩阵$\boldsymbol{M}^{-1}$则能够将坐标由标架B映射到标架A。

8．我们可以将“令几何体自身发生改变”的变换解释为坐标变换，反之亦然。在一些情景中，令物体保持不变，使其在多种坐标系之间进行转换则更为直观。但是，若描述物体的相关参考系产生了变化，物体的坐标也要相应地进行改变。而在另外的一些情景中，我们可能更需要使物体仅在一个坐标系中变换，而不改变其参照的参考系。

3.8　练习

1．设τ:$\mathbb{R}^3 \to \mathbb{R}^3$的定义为$\tau(x, y, z) = (x + y, x - 3, z)$。那么，$\tau$是一种线性变换吗？如果是，求出它的标准矩阵表示。

2．设τ:$\mathbb{R}^3 \to \mathbb{R}^3$的定义为$\tau(x, y, z) = (3x + 4z, 2x - z, x + y + z)$。那么，$\tau$是否是一种线性变换？如果是，求出它的标准矩阵表示。

3．设τ:$\mathbb{R}^3 \to \mathbb{R}^3$是一种线性变换，而且$\tau(1, 0, 0) = (3, 1, 2)$，$\tau(0, 1, 0) = (2, -1, 3)$，$\tau(0, 0, 1) = (4, 0, 2)$。求$\tau(1, 1, 1)$。

4．构建一个缩放矩阵，使物体在x轴方向上放大2倍，在y轴方向上放大−3倍，在z轴方向上保持不变。

5．构建一个旋转矩阵，使物体绕轴(1, 1, 1)旋转$30°$。

6．构建一个平移矩阵，使物体沿x轴正方向平移4个单位，在y轴方向保持不变，沿z轴正方向平移−9个单位。

7．构建一个单独的变换矩阵，首先使物体在x轴方向上放大2倍，在y轴方向上放大−3倍，在z轴上保持不变。接着将物体沿x轴正方向平移4个单位，在y轴上保持不变，沿z轴正方向平移−9个单位。

8．构建一个单独的变换矩阵，首先令物体绕y轴旋转$45°$。接着使之沿x轴正方向平移−2个单位，沿y轴正方向平移5个单位，最后沿z轴正方向平移1个单位。

9．重新计算例3.2，但是这次使其中的正方形在x轴方向上放大1.5倍，在y轴方向上缩小至0.75倍，在z轴上保持不变。最后，绘制出变换前后的几何体，以确定所得到结果是否正确。

10．重新计算例3.3，但是这次将其中的正方形绕y轴顺时针方向旋转$-45°$（即以逆时针方向旋转$45°$）。最后，绘制出变换前后的几何体，验证所得到的结果。

11．重新计算例3.4，此次将该正方形在x轴正方向平移−5个单位，在y轴正方向平移−3.0个单位，沿z轴正方向平移4.0个单位。最后，画出变换前后的几何体，确认所得到结果的正确性。

12．证明$R_n(\boldsymbol{v}) = \cos\theta\boldsymbol{v} + (1-\cos\theta)(\boldsymbol{n}\cdot\boldsymbol{v})\boldsymbol{n} + \sin\theta(\boldsymbol{n}\times\boldsymbol{v})$是一种线性变换。求出它的标准矩阵表示。

13．证明$\boldsymbol{R}_y$中的行向量都是规范正交的。作为拓展，读者也可以将此性质推广到一般的旋转矩阵（绕任意轴旋转的旋转矩阵）中。

14．证明矩阵$\boldsymbol{M}$是正交矩阵，当且仅当$\boldsymbol{M}^T = \boldsymbol{M}^{-1}$。

15．计算：

$$[x,\ y,\ z,\ 1]\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\b_x & b_y & b_z & 1\end{bmatrix}\text{和}\quad [x,\ y,\ z,\ 0]\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\b_x & b_y & b_z & 1\end{bmatrix}$$

平移矩阵是否对点进行了平移操作？又是否平移了向量？为什么对

一个标准位置上的向量坐标进行平移是没有意义的？

16．验证文中给出的缩放矩阵的逆矩阵（见3.1.3节）确实是该缩放矩阵的逆；这可以直接通过矩阵的乘法运算$\boldsymbol{S}\boldsymbol{S}^{-1}=\boldsymbol{S}^{-1}\boldsymbol{S}=\boldsymbol{I}$来加以证明。类似地，证明文中给出的平移矩阵（见3.2.3节）的逆矩阵确实是该平移矩阵的逆，即证明$\boldsymbol{T}\boldsymbol{T}^{-1}=\boldsymbol{T}^{-1}\boldsymbol{T}=\boldsymbol{I}$。

17．假设我们已知标架A和标架B。设$\boldsymbol{p}_A=(1,-2,0)$和$\boldsymbol{q}_A=(1,2,0)$分别表示相对于标架$A$的一个点和一个作用力。另外，设$\boldsymbol{Q}_B=(-6,2,0)$，$\boldsymbol{u}_B=\left(\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0\right)$，$\boldsymbol{v}_B=\left(-\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0\right)$及$\boldsymbol{w}_B=(0,0,1)$描述的是标架$A$的原点与3个坐标轴相对于标架$B$的坐标。构建出把标架$A$的坐标映射为标架$B$的坐标的变换矩阵，并求出$\boldsymbol{p}_B=(x,y,z)$和$\boldsymbol{q}_B=(x,y,z)$的齐次坐标。最后，将变换的过程绘制在图纸上，以验证答案的正确性。

18．利用点来对照向量的线性组合可得到一种仿射组合（affine combination）：$\boldsymbol{p}=a_1\boldsymbol{p}_1+\cdots+a_n\boldsymbol{p}_n$，其中$a_1+\cdots+a_n=1$，且$\boldsymbol{p}_1,\cdots,\boldsymbol{p}_n$都为点。可以将标量系数$a_k$看作是对应“点”的权重，用于描述$\boldsymbol{p}_k$对$\boldsymbol{p}$的影响程度。笼统地讲，$a_k$越接近于1，则$\boldsymbol{p}$越趋近于$\boldsymbol{p}_k$，而负的$a_k$则使$\boldsymbol{p}$与$\boldsymbol{p}_k$“背道而驰”。下一道练习题将更直观地说明这一点。满足上述条件的权值组合也称作点$\boldsymbol{p}$的重心坐标（barycentric coordinate）。最后，请证明仿射组合可以写作一个点与一个向量之和的形式：

$$\boldsymbol{p}=\boldsymbol{p}_1+a_2(\boldsymbol{p}_2-\boldsymbol{p}_1)+\cdots+a_n(\boldsymbol{p}_n-\boldsymbol{p}_1)$$

19．考虑由$\boldsymbol{p}_1=(0,0,0)$，$\boldsymbol{p}_2=(0,1,0)$和$\boldsymbol{p}_3=(2,0,0)$这3个点所构成的三角形。绘制出下列点：

（a）$\frac{1}{3}\boldsymbol{p}_1 + \frac{1}{3}\boldsymbol{p}_2 + \frac{1}{3}\boldsymbol{p}_3$

（b）$0.7\boldsymbol{p}_1 + 0.2\boldsymbol{p}_2 + 0.1\boldsymbol{p}_3$

（c）$0.0\boldsymbol{p}_1 + 0.5\boldsymbol{p}_2 + 0.5\boldsymbol{p}_3$

（d）$-0.2\boldsymbol{p}_1 + 0.6\boldsymbol{p}_2 + 0.6\boldsymbol{p}_3$

（e）$0.6\boldsymbol{p}_1 + 0.5\boldsymbol{p}_2 - 0.1\boldsymbol{p}_3$

（f）$0.8\boldsymbol{p}_1 - 0.3\boldsymbol{p}_2 + 0.5\boldsymbol{p}_3$

小题（a）中的点有什么特别之处吗？若按仿射组合公式用$\boldsymbol{p}_1$，$\boldsymbol{p}_2$和$\boldsymbol{p}_3$三点来表示点$\boldsymbol{p}_2$和点$(1, 0, 0)$，则其对应的重心坐标分别是多少？如果重心坐标中有一分量为负，那么，能推测出此对应点$\boldsymbol{p}$与三角形的位置关系吗？

20．判断一个仿射变换的决定性因素之一便是要满足仿射组合。证明仿射变换$\alpha(\boldsymbol{u})$满足仿射组合，即$\alpha(a_1\boldsymbol{p}_1 + \cdots + a_n\boldsymbol{p}_n) = a_1\alpha(\boldsymbol{p}_1) + \cdots + a_n\alpha(\boldsymbol{p}_n)$，其中$a_1 + \cdots + a_n = 1$。

21．考察图3.16。在计算机图形学里，将坐标由标架A中（正方形$[-1, 1]^2$）映射到标架B中（正方形$[0, 1]^2$，其中的y轴正方向与标架A中的y轴正方向正相反）是一种很常见的坐标变换。证明由标架A到标架B的这种坐标变换为：

$$[x,\ y,\ 0,\ 1]\begin{bmatrix}0.5 & 0 & 0 & 0\\ 0 & -0.5 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0.5 & 0.5 & 0 & 1\end{bmatrix}=[x',\ y',\ 0,\ 1]$$

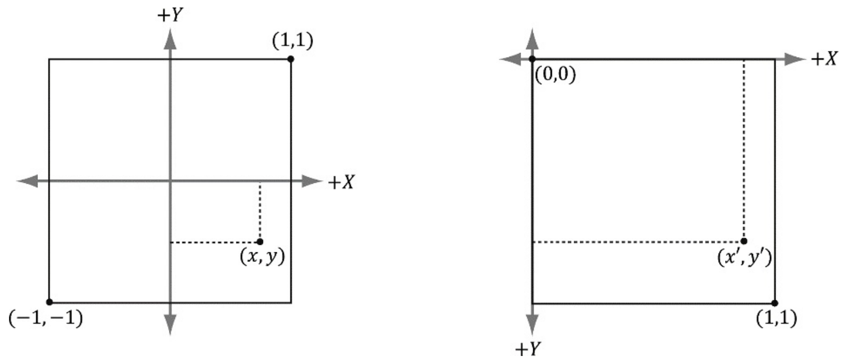


图3.16　将标架A中（正方形$[-1,1]^2$）的坐标映射到标架B（正方形$[0,1]^2$，其中的y轴正方向与标架A中y轴正方向刚好相反）的坐标变换

22．在第2章曾提到：在线性变换下，行列式与（n维平行多面体）体积（面积）的变化有关。请求出缩放矩阵的行列式，并从体积变化的角度来对此进行解释。

23．思考将正方形扭曲为平行四边形的变换τ：

$$\tau(x,\ y)=(3x+y,\ x+2y)$$

求出此变换的标准矩阵表示，并证明变换矩阵的行列式与由$\tau(\boldsymbol{i})$和$\tau(\boldsymbol{j})$张成的平行四边形面积相等（参考图3.17）。

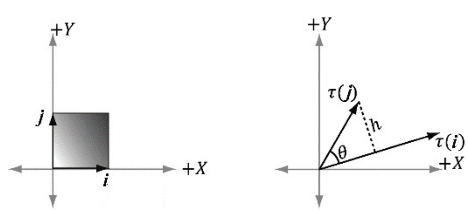


图3.17　将正方形映射为平行四边形的变换

24．证明令物体绕y轴旋转的变换矩阵行列式为1。根据之前的习题，解释为什么此值为1。作为进一步拓展，读者也可以证明：一般旋转矩阵（令物体绕任意轴旋转的矩阵）的行列式皆为1。

25．我们可以将任意的旋转矩阵看作是行列式为1的正交矩阵。结合图3.7与习题24来加以验证便会发现这一点。旋转后的基向量$\tau(\boldsymbol{i})$、$\tau(\boldsymbol{j})$和$\tau(\boldsymbol{k})$都为单位长度，且互相正交；再者，由于旋转变换不会改变物体的大小，因此旋转矩阵的行列式就应当为1。请证明两个旋转矩阵的乘积$\boldsymbol{R}_1\boldsymbol{R}_2=\boldsymbol{R}$也是旋转矩阵。即证明$\boldsymbol{R}\boldsymbol{R}^T=\boldsymbol{R}^T\boldsymbol{R}=\boldsymbol{I}$（以此证明$\boldsymbol{R}$是正交矩阵），以及$\det\boldsymbol{R}=1$。

26．证明旋转矩阵$\boldsymbol{R}$具有下列性质：

（a）$(\boldsymbol{uR})\cdot(\boldsymbol{vR})=\boldsymbol{u}\cdot\boldsymbol{v}$　　点积不变性

（b）$||\boldsymbol{uR}||=||\boldsymbol{u}||$　　保长性

（c）$\theta(\boldsymbol{uR},\boldsymbol{vR})=\theta(\boldsymbol{u},\boldsymbol{v})$　　保角性，其中$\theta(\boldsymbol{x},\boldsymbol{y})$

计算的是$\boldsymbol{x}$与$\boldsymbol{y}$之间的夹角：

$$\theta(\boldsymbol{x},\boldsymbol{y})=\arccos\frac{\boldsymbol{x}\cdot\boldsymbol{y}}{||\boldsymbol{x}||||\boldsymbol{y}||}$$

解释这些性质对于旋转变换的意义。

27．求出一个兼有缩放、旋转和平移操作的矩阵，通过它将始于点$\boldsymbol{p}=(0,0,0)$、终于点$\boldsymbol{q}=(0,0,1)$的线段变换为始于点$(3,1,2)$、平行于向量

$(1,1,1)$且长度为2的线段。

28．假设有一中心位于坐标(x, y, z)处的立方体。若定义原点为缩放变换的参考点，那么，对立方体（注意，它的中心此时并非位于原点）进行缩放处理便会产生使之平移的“副作用”，这在某些应用情景中是我们所不愿看到的。因此，请求出一种使该立方体相对于其中心点进行缩放的变换。

提示

首先通过坐标变换将立方体转换至原点位于其中心的立方体坐标系，对立方体进行缩放处理后，再将其变换回起始坐标系，如图3.18所示。

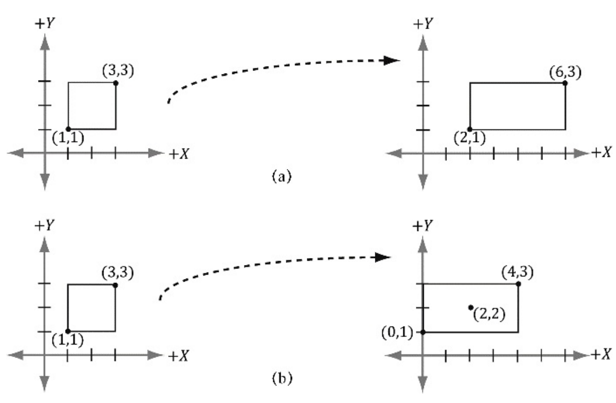


图3.18　长方体的缩放变换

（a）令正方形相对于原点在x轴方向上放大两倍会导致变换后的矩形发生平移

（b）使正方形相对于自身中心点在x轴方向上放大两倍则不会使变换后的矩形发生平移（即变换后的矩形中心点仍位于变换前正方形的起始中心点处）

---

[1] 由此可见，这里采用的是右手坐标系，下面的示例亦是如此，但Direct3D中实际采用的是左手坐标系。

[2] 原文为“composition of transformations”，也有译作变换的组合等。

[3] 原文为change of coordinate transformation，但这里译作坐标系变换或许更贴切。

第二部分　Direct3D基础

在这一部分中，我们将学习贯穿本书后续内容的Direct3D基础概念以及技术。掌握了这些基本功后，我们就能写出更加有趣的应用程序。以下是本部分各章的简介。

第**4**章**“Direct3D**的初始化**”**这一章将带领读者进一步理解Direct3D并学习如何对它进行初始化，为后续的3D绘图工作打好基础。另外，也会介绍一些Direct3D的基本技术主题，例如表面[1]、像素格式（pixel format）、页面翻转（page flipping）、深度缓冲（depth buffering）和多重采样（multisampling）。我们还会学习用性能计数器（performance counter）度量时间，用于统计每秒中所渲染的帧数。除此之外，还会给出一些有关Direct3D的调试小窍门。最后，我们会开发和使用属于自己的应用程序框架——当然，这并不是指SDK框架。

第**5**章**“**渲染流水线**”**在这篇幅较长一章里将对渲染流水线（rendering pipeline）进行全面地讲解。渲染流水线是基于虚拟摄像机（virtual camera）的视角来进行观察，并据此生成场景2D图像的一系列步骤。在此，我们将学习定义3D场景、控制虚拟摄像机，以及将3D几何体投影至一个2D图像的平面中。

第**6**章**“**利用**Direct3D**绘制几何体**”**这一章将关注：定义3D几何体、配置渲染流水线、创建顶点着色器（vertex shader）和像素着色器

（pixel shader），以及向渲染流水线提交用于绘制的几何体等操作相关的Direct3D API接口与方法。结束本章的学习后，我们将能够绘制一个3D立方体并从不同的角度来观赏它。

第**7**章**“**利用**Direct3D**绘制几何体（续）**”**这一章将介绍本书后续要用到的几种绘制模式。借助优化CPU与GPU之间的工作负载平衡，引出重新组织绘制物体的渲染流程这一主题。本章最后将展示怎样渲染出更为复杂的物体，如栅格、球体、立柱乃至模拟动态的波浪。

第**8**章**“**光照**”**这一章展示了光源的创建过程，并定义了光与不同材质表面之间的交互。在此，我们还特别演示了如何用顶点着色器和像素着色器来实现平行光光源（directional lights）、点光源（point lights）和聚光灯光源（spot lights）。

第**9**章**“**纹理贴图**”**这一章描述了纹理贴图（texture mapping），这是一种通过将2D图像数据映射到3D图元上、继而使场景更加真实的技术。例如，运用纹理贴图，我们就可以把2D砖块的图片应用到一个3D立方体的表面，以此来模拟砖块。其他关键的纹理主题也将在这一章讲授，其中包括纹理平铺（texture tiling）和动态纹理变换（animated texture transformation）。

第**10**章**“**混合**”**利用混合（blending）技术，我们便可以实现许多如透明度（transparency）这样的特效。另外，我们将在这一章探讨HLSL内置的裁剪函数（clip），通过它便可以从可视的图像中掩盖住指定的部分，比如说，此函数可用于绘制铁丝网和门等物体。另外，本章还会

展示如何实现雾的效果。

第11章“模板”这一章介绍了模板缓冲区（stencil buffer），顾名思义，它就像是一块“模板”，允许我们阻止特定像素的绘制操作。而且，这种遮罩像素的技术在各种情况下都适用。为了使读者充分理解本章的主题，我们将深入研究用模板缓冲区实现平面反射（planar reflection）和平面阴影（planar shadow）的方法。

第12章“几何着色器”这一章展示了如何编写几何着色器（geometry shader）。几何着色器比较特殊，因为它可以创建和销毁整个几何图元（geometric primitive）。几何着色器的常见应用场合有公告牌（billboard）、毛发渲染（fur rendeing）、细分（subdivision）和粒子系统（particle system）。另外，这一章还阐释了图元ID和纹理数组（texture array）等概念。

第13章“计算着色器”计算着色器（compute shader）是一种可编程着色器。Direct3D提供的计算着色器并非直属于渲染流水线。通过它即可将图形处理器（GPU）应用于通用计算（general purpose computation）。例如，一款图像应用软件可以利用计算着色器，实现GPU对图像处理算法的加速。由于计算着色器是Direct3D的一部分，所以它的读写操作都依赖于Direct3D的资源。这样一来，我们就可以直接把计算结果整合到渲染流水线中。因此，除了通用计算以外，计算着色器依然可用于3D渲染。

第14章“曲面细分阶段”这一章将探索渲染流水线的曲面细分阶段

（tessellation stage）。利用这个阶段中的镶嵌技术便能够将几何体细分为更小的三角形，接着再以某种方式对新生成的顶点进行偏移。其中，增加三角形数量的动机是使网格增添细节。本章将详解这项技术的工作原理，我们会展示怎样根据四边形面片的观察距离来对它进行镶嵌化处理，也将演示如何来渲染三次贝塞尔四边形面片的表面。

---

[1]　原文为surface。在Direct3D中，表面（surface，不要与“物体表面”混淆）这一术语表示显存（尽管表面也可位于系统内存，但这里通常指代的是显存端）中的一块线性区域。这个词在DirectX 9时期比较常见，但在DirectX 12文档中很少被提及，可认为是各种缓冲区、纹理等2D资源的一种低层抽象或旧代名词。

第4章　Direct3D的初始化

为了理解Direct3D的初始化过程，4.1 节和4.2 节将用来讲述我们需要熟悉的一些相关知识，如Direct3D中的数据类型和基础的图形学概念。接下来，我们会继续深入Direct3D初始化步骤的具体细节。随后我们会留出一些篇幅来探讨实时图形应用所需要的精确计时和时间度量。最后，我们将研究示例的框架代码，它为本书后续的演示程序提供了统一的接口。

学习目标：

1．了解Direct3D在3D编程中相对于硬件所扮演的角色。

2．理解组件对象模型COM在Direct3D中起到的作用。

3．掌握基础的图形学概念，例如2D图像的存储方式、页面翻转、深度缓冲、多重采样以及CPU与GPU之间的交互。

4．学习使用性能计数器函数，以此读取高精度计时器的数值。

5．了解Direct3D的初始化过程。

6．熟悉本书应用程序框架的整体结构，我们在后续的演示程序中总会见到它的身影。

4.1　预备知识

要学习Direct3D的初始化流程，我们还需要了解一些基本的图形学概念以及Direct3D中常用数据类型的相关知识。本节将着重介绍这些内容，以防在后面讲解Direct3D的初始化流程时被这些细枝末节喧宾夺主。

4.1.1　Direct3D 12概述

通过Direct3D这种底层图形应用程序编程接口（Application Programming Interface，API），即可在在应用程序中对图形处理器（Graphics Processing Unit，GPU）进行控制和编程。我们能够借此以硬件加速的方式渲染出虚拟的3D场景。例如，若要向GPU提交一个清除某渲染目标[1]（如清屏）的命令，我们就可以调用Direct3D中的`ID3D12GraphicsCommandList::ClearRenderTargetView`方法[2]。随后，Direct3D层和硬件驱动会协作将此Direct3D命令转换为系统中GPU可以执行的本地机器指令。这就是说，只要GPU支持当前所用的Direct3D版本，我们就无须再考虑它的具体规格和硬件控制层面的实现细节。为此，GPU的生产厂商如NVIDIA、Intel和AMD等公司就必须与Direct3D团队一同合作，为用户提供与Direct3D设备相兼容的驱动。

除了添加一些新的渲染特性以外，Direct3D 12经重新设计已焕然一新，较之上一个版本的主要改变在于其性能优化方面在大大减少了CPU

开销的同时，又改进了对多线程的支持。为了达到这些性能目标，Direct3D 12的API较Direct3D 11更偏于底层。另外，API抽象程度的降低使它更趋于具体化，与现代GPU的构架也更为契合，因此也就促使开发者要付出比昔日更多的努力。当然，使用这种更复杂的API所得到的回报是：性能的提升。

4.1.2 组件对象模型

组件对象模型（Component Object Model，COM）是一种令DirectX不受编程语言束缚，并且使之向后兼容的技术。我们通常将COM对象视为一种接口，但考虑当前编程的目的，遂将它当作一个C++类来使用。用C++语言编写DirectX程序时，COM帮我们隐藏了大量底层细节。我们只需知道：要获取指向某COM接口的指针，需借助特定函数或另一COM接口的方法——而不是用C++语言中的关键字`new`去创建一个COM接口。另外，COM对象会统计其引用次数；因此，在使用完某接口时，我们便应调用它的`Release`方法（COM接口的所有功能都是从`IUnknown`这个COM接口继承而来的，包括`Release`方法在内），而不是用`delete`来删除——当COM对象的引用计数为0时，它将自行释放自己所占用的内存。

为了辅助用户管理COM对象的生命周期，Windows运行时库（Windows Runtime Library，WRL）专门为此提供了`Microsoft::WRL::ComPtr`类（`#include <wrl.h>`），我们可以把它当作是COM对象的智能指针。当一个`ComPtr`实例超出作用域范围时，

它便会自动调用相应COM对象的`Release`方法，继而省掉了我们手动调用的麻烦。本书中常用的3个`ComPtr`方法如下。

1．`Get`：返回一个指向此底层COM接口的指针。此方法常用于把原始的COM接口指针作为参数传递给函数。例如：

```
ComPtr<ID3D12RootSignature> mRootSignature;
...
// SetGraphicsRootSignature需要获取ID3D12RootSignature*类型的参数
mCommandList->SetGraphicsRootSignature(mRootSignature.Get());
```

2．`GetAddressOf`：返回指向此底层COM接口指针的地址。凭此方法即可利用函数参数返回COM接口的指针。例如：

```
ComPtr<ID3D12CommandAllocator> mDirectCmdListAlloc;
...
ThrowIfFailed(md3dDevice->CreateCommandAllocator(
  D3D12_COMMAND_LIST_TYPE_DIRECT,
  mDirectCmdListAlloc.GetAddressOf()));
```

3．`Reset`：将此`ComPtr`实例设置为`nullptr`释放与之相关的所有引用（同时减少其底层COM接口的引用计数）。此方法的功能与将`ComPtr`目标实例赋值为`nullptr`的效果相同。

当然，与COM有关的知识并不止于此，但是对有效地使用DirectX来说足矣。

注意 Note

COM接口都以大写字母“I”作为开头。例如，表示命令列表的COM接口为ID3D12GraphicsCommandList。

4.1.3　纹理格式

2D纹理（2D texture）是一种由数据元素构成的矩阵（可将此“矩阵”看作2D数组）。它的用途之一是存储2D图像数据，在这种情况下，纹理中每个元素存储的都是一个像素[3]的颜色。然而，纹理的用处并非仅此而已。例如，有种称作法线贴图（normal mapping）的高级技术，其纹理内的每个元素存储的就是一个3D向量而不是颜色信息。因此，尽管纹理给人的第一印象通常是用来存储图像数据，但其实际用途却十分广泛。简单来讲，1D、2D、3D纹理就相当于特定数据元素所构成1D、2D、3D数组。但随着后续章节中对纹理讨论的逐渐深入，我们便会知道，纹理其实还不只是像“数据数组”那样简单。它们可能还具有多种mipmap层级[4]，而GPU则会据此对它们进行特殊的处理，例如运用过滤器（filter）和进行多重采样（multisample）。另外，并不是任意类型的数据元素都能用于组成纹理，它只能存储DXGI_FORMAT枚举类型中描述的特定格式的数据元素。下面是一些相关的格式示例：

1．DXGI_FORMAT_R32G32B32_FLOAT：每个元素由3个32位浮点数分量构成。

2．DXGI_FORMAT_R16G16B16A16_UNORM：每个元素由4个16位分

量构成，每个分量都被映射到 [0, 1] 区间。

3．`DXGI_FORMAT_R32G32_UINT`：每个元素由2个32位无符号整数分量构成。

4．`DXGI_FORMAT_R8G8B8A8_UNORM`：每个元素由4个8位无符号分量构成，每个分量都被映射到 [0, 1] 区间。

5．`DXGI_FORMAT_R8G8B8A8_SNORM`：每个元素由4个8位有符号分量构成，每个分量都被映射到 [−1, 1] 区间。

6．`DXGI_FORMAT_R8G8B8A8_SINT`：每个元素由4个8位有符号整数分量构成，每个分量都被映射到 [−128, 127] 区间。

7．`DXGI_FORMAT_R8G8B8A8_UINT`：每个元素由4个8位无符号整数分量构成，每个分量都被映射到 [0, 255] 区间。

注意，大写字母R、G、B、A分别表示红色（red）、绿色（green）、蓝色（blue）和alpha。所有的颜色都是由红、绿、蓝三基色[5]组合而成（例如，红色和绿色混合成黄色）。alpha通道（或称为alpha分量）则通常用于控制透明度。然而，正如前文所述，尽管格式名称在字面上指示的是颜色和alpha值，但纹理存储的却不一定是颜色信息。例如，格式

```
DXGI_FORMAT_R32G32B32_FLOAT
```

中含有3个浮点数分量，因此可以利用坐标格式为浮点数的方式存

储任意3D向量。除此之外，亦有无类型（typeless）格式的纹理，我们仅用它来预留内存，待纹理被绑定到渲染流水线（rendering pipeline，详见第5章）之后，再具体解释它的数据类型（有点像C++语言里的强制转换[6]）。例如，下面的无类型格式保留的是由4个16位分量组成的元素，但并没有指出数据的具体类型（例如，是整数、浮点数还是无符号整数？）：

```
DXGI_FORMAT_R16G16B16A16_TYPELESS
```

我们将在第6章中看到DXGI_FORMAT枚举类型也可用于描述顶点以及索引的数据格式。

4.1.4　交换链和页面翻转

为了避免动画中出现画面闪烁的现象，最好将动画帧完整地绘制在一种称为后台缓冲区的离屏（off-screen，即不可直接呈现在显示设备上之意）纹理内。只要将指定动画帧的整个场景绘到后台缓冲区中，它就会以一个完整的帧画面展现在屏幕上；依照此法，观者便不会察觉出帧的绘制过程——而只会观赏到完整的动画帧。为此，需要利用由硬件管理的两种纹理缓冲区：即所谓的前台缓冲区（front buffer）和后台缓冲区（back buffer）。前台缓冲区存储的是当前显示在屏幕上的图像数据，而动画的下一帧则被绘制在后台缓冲区里。当后台缓冲区中的动画帧绘制完成之后，两种缓冲区的角色互换：后台缓冲区变为前台缓冲区呈现新一帧的画面，而前台缓冲区则为了展示动画的下一帧转为后台缓冲区，等待填充数据。前后台缓冲的这种互换操作称为呈现

（presenting，亦有译作提交、显示等）。呈现是一种高效的操作，只需交换指向当前前台缓冲区和后台缓冲区的两个指针即可实现。图4.1详细地解释了这个过程。

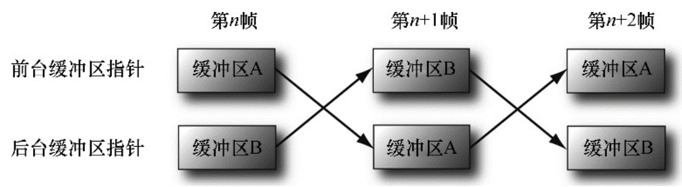


图4.1　对于第n帧来讲，当前显示的是缓冲区A中的内容，我们将把下一帧的数据渲染到此时的后台缓冲区B内。一旦后台缓冲区绘制完毕，两个缓冲区的指针将互换，即缓冲区B将变成前台缓冲区，而缓冲区A则成为新的后台缓冲区。接下来，我们会把下一帧的内容渲染到缓冲区A中。待后台缓冲区（即此时的缓冲区A）完成绘制，两个缓冲区的指针再次互换，即在第$n+2$帧中，缓冲区A重新成为前台缓冲区，缓冲区B则再次客串后台缓冲区

前台缓冲区和后台缓冲区构成了交换链（swap chain），在Direct3D中用`IDXGISwapChain`接口来表示。这个接口不仅存储了前台缓冲区和后台缓冲区两种纹理，而且还提供了修改缓冲区大小（`IDXGISwapChain::ResizeBuffers`）和呈现缓冲区内容（`IDXGISwapChain::Present`）的方法。

使用两个缓冲区（前台和后台）的情况称为双缓冲（double buffering，亦有译作双重缓冲、双倍缓冲等）。当然，也可以运用更多的缓冲区。例如，使用3个缓冲区就叫作三重缓冲（triple buffering，亦有译作三倍缓冲等[7]）。对于一般的应用来说，使用两个缓冲区就足够了。

注意 Note

尽管后台缓冲区是一个纹理（因而构成纹理的基本元素又称纹素，texel），但我们仍常将其组成元素称为像素，因为就后台缓冲区这种情况而言，它所存储的内容是颜色信息。即便纹理中存储的不是颜色信息，大家有时也称纹理的元素为像素（如“法线图中的像素”）[8]。

4.1.5　深度缓冲

深度缓冲区（depth buffer）这种纹理资源存储的并非图像数据，而是特定像素的深度信息。深度值的范围为0.0～1.0。0.0代表观察者在视锥体（view frustum，亦有译作视域体、视景体、视截体或视体等，意即观察者能看到的空间范围，形如从四棱锥中截取的四棱台，常称该形为平截头体（frustum，见图4.3，后文亦有详述））中能看到离自己最近的物体，1.0则代表观察者在视锥体中能看到离自己最远的物体。深度缓冲区中的元素与后台缓冲区内的像素呈一一对应关系（即后台缓冲区中第i行第j列的元素对应于深度缓冲区内第i行第j列的元素）。所以，如果后台缓冲区的分辨率为1280×1024，那么深度缓冲区中就应当有1280×1024个深度元素。

图4.2展现了一个简单的场景，其中，后侧物体的局部区域被其他物体所遮挡。为了确定不同物体间的像素前后顺序，Direct3D采用了一种叫作深度缓冲（depth buffering）或**z**缓冲（z-buffering，其中的z指z坐

标）的技术。这里要着重强调一个细节：若使用了深度缓冲，则物体的绘制顺序也就变得无关紧要了。

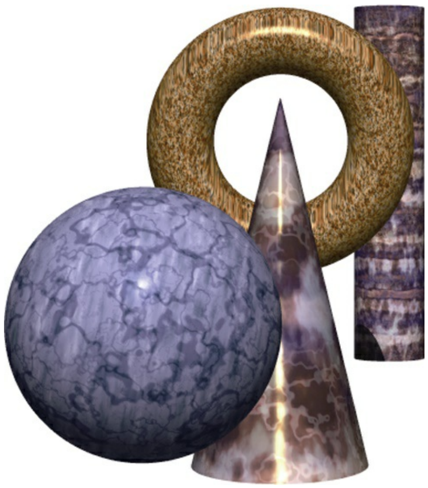


图4.2　一组互有遮挡的物体

注意 Note

针对深度问题的处理，有读者可能会提出：不妨将场景中的物体按由远及近的顺序来绘制。照这种方式，远处的物体就会被近处的物体所覆盖，并渲染出正确效果。这其实就是画家绘制景物的方法。然而，这种方法其实也有它自己的缺陷——绘制过程中，不仅需要对大量的数据按从后至前的绘制顺序进行排序，而且还涉及几何体相交的问题。较之这种处理方式，图形硬件还特别提供了深度缓冲供开发者自由使用，对此，我们何乐而不为呢？

为了对深度缓冲的工作原理有更深入的了解，让我们来看一个例子。如图4.3所示，图中展示了观察者看到的立体空间，以及该立体空间的2D侧视图。从图中可以看到，有3种不同物体的像素都争着渲染在观察窗口内的像素P上（我们当然知道离观察者最近的像素会被渲染到像素P，因为它会遮住后面所有物体的对应像素。但是计算机却不知道）。在开始渲染之前，后台缓冲区会被清理为默认颜色，深度缓冲区也将被清除为默认值——通常为1.0（即像素能够取到的最远深度值）。现在，假设这些物体的渲染顺序依次为圆柱体→球体→圆锥体。下面的列表总结了像素P和它对应的深度值d按物体的绘制顺序依次更新的过程。类似的处理流程也发生在其他像素上。

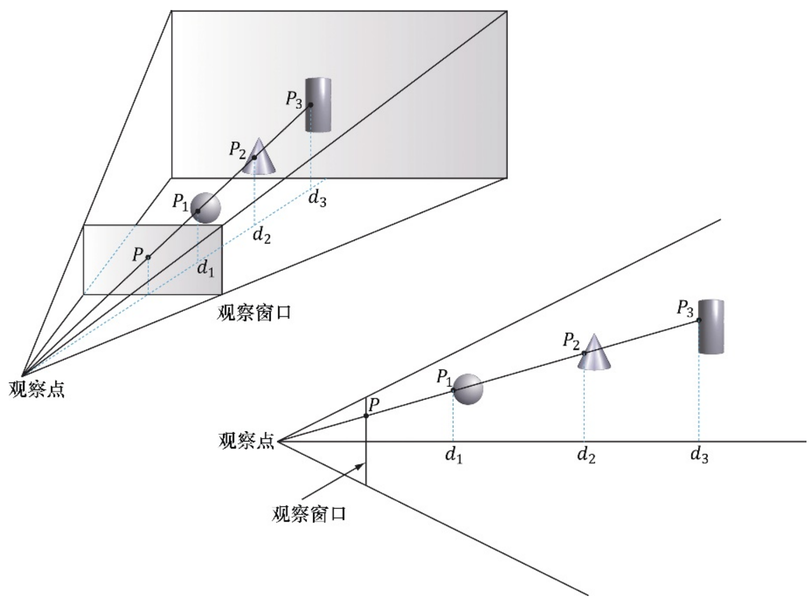


图4.3　为3D场景生成位于观察窗口内对应2D图像（后台缓冲区）的过程。可以看出，图中有3个不同的像素争着投影在像素P上。直觉告诉我们，像素P_1理应被写到P内，因为它离观察者最近，会遮挡住其后的另外两个同位像素。深度缓冲区算法为计算机确定像素P提供了一种机械化的处理流程。注意，这里讨论的深度值是相对于被观测的3D场景来说的，而深度缓冲区中所存的实际深度值范围为$[0.0, 1.0]$

操作步骤	P	d	步骤叙述
清除缓冲区操作	黑色	1.0	对像素及其对应的深度元素进行初始化
绘制圆柱体	P_3	d_3	因为$d_3 \leqslant d = 1.0$，深度测试通过，更新缓冲区，使$P = P_3$，$d = d_3$
绘制球体	P_1	d_1	因为$d_1 \leqslant d = d_3$，深度测试通过，更新缓冲区，使$P = P_1$，$d = d_1$
绘制圆锥体	P_1	d_2	因为$d_2 > d = d_1$，深度测试失败，不更新缓冲区

可以看出，只有找到具有更小深度值的像素，才会对观察窗口内的像素及其位于深度缓冲区中的对应深度值进行更新。按照这种方法逐步处理，待完成所有的比较和更新工作后，最终得到渲染的即为距离观察者最近的像素（如果读者仍对此法感到将信将疑，可以尝试更改几何体的绘制顺序，再依上述表格的方法一步步重新推导）。

总而言之，深度缓冲技术的原理是计算每个像素的深度值，并执行深度测试（depth test）。而深度测试则用于对竞争写入后台缓冲区中同一像素的多个像素深度值进行比较。具有最小深度值的像素（说明该像素离观察者最近）会获得最终的胜利，它将被写入后台缓冲区中。这样做也是合乎情理的，因为离观察者较近的像素无疑会遮挡其后面的像素。

深度缓冲区也是一种纹理，所以一定要用明确的数据格式来创建它。深度缓冲可用的格式包括以下几种。

1．DXGI_FORMAT_D32_FLOAT_S8X24_UINT：该格式共占用64位，取其中的32位指定一个浮点型深度缓冲区，另有8位（无符号整数）分配给模板缓冲区（stencil buffer），并将该元素映射到[0, 255]区间，剩下的24位仅用于填充对齐（padding）不作他用。

2．DXGI_FORMAT_D32_FLOAT：指定一个32位浮点型深度缓冲区。

3．DXGI_FORMAT_D24_UNORM_S8_UINT：指定一个无符号24位深度缓冲区，并将该元素映射到[0, 1]区间。另有8位（无符号整型）分配给模板缓冲区，将此元素映射到[0, 255]区间。

4．DXGI_FORMAT_D16_UNORM：指定一个无符号16位深度缓冲区，把该元素映射到[0, 1]区间。

注意 Note

一个应用程序不一定要用到模板缓冲区。但一经使用，则深度缓冲区将总是与模板缓冲区如影随形，共同进退。例如，32位格式

```
DXGI_FORMAT_D24_UNORM_S8_UINT
```

使用24位作为深度缓冲区，其他8位作为模板缓冲区。出于这个原因，深度缓冲区叫作深度/模板缓冲区更为得体。模板缓冲区的运用是

更高级的主题，在第11章中会有相应的介绍。

4.1.6　资源与描述符

在渲染处理的过程中，GPU可能会对资源进行读（例如，从描述物体表面样貌的纹理或者存有3D场景中几何体位置信息的缓冲区中读取数据）和写（例如，向后台缓冲区或深度/模板缓冲区写入数据）两种操作。在发出绘制命令之前，我们需要将与本次绘制调用（draw call）相关的资源绑定（bind或称链接，link）到渲染流水线上。部分资源可能在每次绘制调用时都会有所变化，所以我们也就要每次按需更新绑定。但是，GPU资源并非直接与渲染流水线相绑定，而是要通过一种名为描述符（descriptor）的对象来对它间接引用，我们可以把描述符视为一种对送往GPU的资源进行描述的轻量级结构。从本质上来讲，它实际上即为一个中间层；若指定了资源描述符，GPU将既能获得实际的资源数据，也能了解到资源的必要信息。因此，我们将把绘制调用需要引用的资源，通过指定描述符的方式绑定到渲染流水线。

为什么我们要额外使用描述符这个中间层呢？究其原因，GPU资源实质都是一些普通的内存块。由于资源的这种通用性，它们便能被设置到渲染流水线的不同阶段供其使用。一个常见的例子是先把纹理用作渲染目标（即Direct3D的绘制到纹理技术），随后再将该纹理作为一个着色器资源（即此纹理会经采样[9]而用作着色器的输入数据）。不管是充

当渲染目标、深度/模板缓冲区还是着色器资源等角色，仅靠资源本身是无法体现出来的。而且，我们有时也许只希望将资源中的部分数据绑定至渲染流水线，但如何从整个资源中将它们选取出来呢？再者，创建一个资源可能用的是无类型格式，这样的话，GPU甚至不会知道这个资源的具体格式。

解决上述问题就是引入描述符的原因。除了指定资源数据，描述符还会为GPU解释资源：它们会告知Direct3D某个资源将如何使用（即此资源将被绑定在流水线的哪个阶段上），而且我们可借助描述符来指定欲绑定资源中的局部数据。这就是说，如果某个资源在创建的时候采用了无类型格式，那么我们就必须在为它创建描述符时指明其具体类型。

注意 Note

视图（view）与描述符（descriptor）是同义词。“视图”虽是Direct3D先前版本里的常用术语，但它仍然沿用在Direct3D 12的部分API中。在本书里，两者交替使用，例如，“常量缓冲区视图（constant buffer view）”与“常量缓冲区描述符（constant buffer descriptor）”表达的是同一事物。

每个描述符都有一种具体类型，此类型指明了资源的具体作用。本书常用的描述符如下。

1．CBV/SRV/UAV描述符分别表示的是常量缓冲区视图（constant buffer view）、着色器资源视图（shader resource view）和无序访问视图（unordered access view）这3种资源。

2．采样器（sampler，亦有译为取样器）描述符表示的是采样器资源（用于纹理贴图）。

3．RTV描述符表示的是渲染目标视图资源（render target view）。

4．DSV描述符表示的是深度/模板视图资源（depth/stencil view）。

描述符堆（descriptor heap）中存有一系列描述符（可将其看作是描述符数组），本质上是存放用户程序中某种特定类型描述符的一块内存。我们需要为每一种类型的描述符都创建出单独的描述符堆。另外，也可以为同一种描述符类型创建出多个描述符堆。

我们能用多个描述符来引用同一个资源。例如，可以通过多个描述符来引用同一个资源中不同的局部数据。而且，前文曾提到过，一种资源可以绑定到渲染流水线的不同阶段。因此，对于每个阶段都需要设置独立的描述符。例如，当一个纹理需要被用作渲染目标与着色器资源时，我们就要为它分别创建两个描述符：一个RTV描述符和一个SRV描述符。类似地，如果以无类型格式创建了一个资源，又希望该纹理中的元素可以根据需求当作浮点值或整数值来使用，那么就需要为它分别创建两个描述符：一个指定为浮点格式，另一个指定为整数格式。

创建描述符的最佳时机为初始化期间。由于在此过程中需要执行一

些类型的检测和验证工作，所以最好不要在运行时（runtime）才创建描述符。

注意 Note

2009年8月的SDK文档写到：“所谓创建一个完整类型的资源，即在资源创建的伊始就确定了它的具体格式。这将使运行时的访问操作得到优化 [·· ·· ··]。”因此，当确实需要用到无类型资源所带来的灵活性时（即根据不同的视图对同一种数据进行多种不同解释的能力），再以这种方式来创建资源，否则应创建完整类型的资源。

4.1.7 多重采样技术的原理

由于屏幕中显示的像素不可能是无穷小的[10]，所以并不是任意一条直线都能在显示器上“平滑”而完美地呈现出来。图4.4所示的，即为以像素矩阵（matrix of pixels，可以理解为“像素2D数组”）逼近直线的方法所产生的“阶梯”（aliasing，锯齿状走样）效果。类似地，显示器中呈现的三角形之边也存在着不同程度的锯齿效应。



图4.4　我们能够明显地看出上面的直线存在锯齿效应（这种阶梯状的效果是由以像素矩阵来表示直线所导致的）。下面的反走样直线，则是通过对每个像素周围的像素进行采样，并生成其最终的颜色而得到的。利用这种方法能够在一定程度上缓解阶梯效应并得到更加平滑的图像

通过提高显示器的分辨率就能够缩小像素的大小[11]，继而使上述问题得到显著地改善，使阶梯效应在很大程度上不易被用户所察觉。

在不能提升显示器分辨率，或在显示器分辨率受限的情况下，我们就可以运用各种反走样（antialiasing，也有译作抗锯齿、反锯齿、反失真等）技术。有一种名为超级采样（supersampling，可简记作SSAA，即Super Sample Anti-Aliasing）的反走样技术，它使用4倍于屏幕分辨率大小的后台缓冲区和深度缓冲区。3D场景将以这种更大的分辨率渲染到后台缓冲区中。当数据要从后台缓冲区调往屏幕显示的时候，会将后台缓冲区按4个像素一组进行解析（resolve，或称降采样，downsample。把放大的采样点数降低回原采样点数）：每组用求平均值的方法得到一种相对平滑的像素颜色。因此，超级采样实际上是通过软件的方式提升了画面的分辨率。

超级采样是一种开销高昂的操作，因为它将像素的处理数量和占用的内存大小都增加到之前的4倍。对此，Direct3D还支持一种在性能与效果等方面都较为折中的反走样技术，叫作多重采样（multisampling，可简记作MSAA，即MultiSample Anti-Aliasing）。这种技术通过跨子像素[12]共享一些计算信息，从而使它比超级采样的开销更低。现假设采用4X多重采样（即每个像素中都有4个子像素），并同样使用4倍于屏幕分辨率的后台缓冲区和深度缓冲区。值得注意的是，这种技术并不需要对每一个子像素都进行计算，而是仅计算一次像素中心处的颜色，再

基于可视性（每个子像素经深度/模板测试的结果）和覆盖性（子像素的中心在多边形的里面还是外面？）将得到的颜色信息分享给其子像素[13]。图4.5展示了一个多重采样的相关实例。

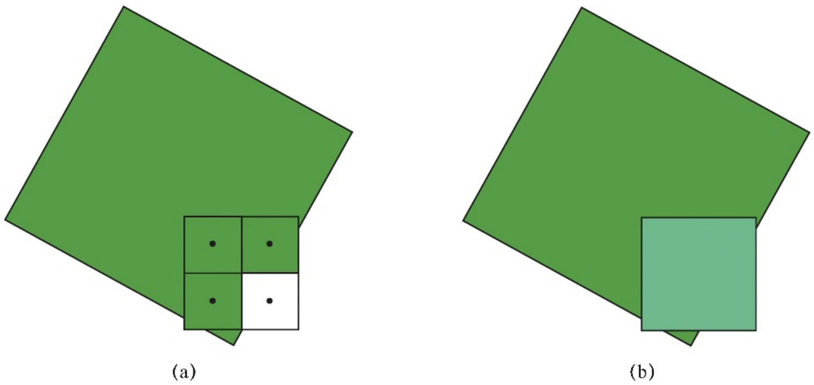


图4.5　现在我们来考虑将一个位于多边形边沿上的像素进行多重采样处理。图a中，我们采集该像素中心的
绿色数据，并将它存于此多边形所覆盖的3个可见子像素中。由于第4个子像素不在该多边形的范围之内，
因此并不将它更新为绿色，而是令其继续保持之前几何体绘制时所计算出的颜色或是清除缓冲区时所得到的
颜色。图b中，为了计算降采样的像素颜色，通过对4个子像素（3个绿色像素以及1个白色像素）
求取平均值的方式，获得多边形边沿上的一种浅绿色。由于抗锯齿方法有效地缓解了
多边形边沿处的阶梯效应，因此图像看起来更为平滑[14]

注意　Note

超级采样和多重采样的关键区别是显而易见的。对于超级采样来说，图像颜色要根据每一个子像素来计算，因此每个子像素都可能各具不同的颜色。而以多重采样的方式（见图4.5b）来求取图像颜色时，每

个像素只需计算一次，最后，再将得到的颜色数据复制到多边形覆盖的所有可见子像素之中。由于计算图像颜色是图形流水线中开销最大的步骤之一，所以用多重采样来代替超级采样对节省资源而言意义非凡。但是话说回来，超级采样的精准度确实更高一筹。

图4.5所示的是一种将每个像素都以均匀栅格划分为4个子像素的反锯齿采样模式。实际上，每家硬件厂商所采用的模式（即选定的子像素位置，可以说决定了采样的位置）可能会各不相同，而Direct3D也并没有定义子像素的具体布局。在各种特定的情况下，不同的布局模式各有千秋。

4.1.8　利用Direct3D进行多重采样

在接下来的小节中，我们要学习填写DXGI_SAMPLE_DESC结构体。该结构体中有两个成员，其定义如下：

```
typedef struct DXGI_SAMPLE_DESC
{
  UINT Count;
  UINT Quality;
} DXGI_SAMPLE_DESC;
```

Count成员指定了每个像素的采样次数，Quality成员则用于指示用户期望的图像质量级别（对于不同的硬件生产商而言，“质量级别”的意义可能千差万别）。采样数量越多或质量级别越高，其渲染操作的代

价也就会愈发高昂，所以需要在质量与速度之间做出利弊权衡。至于质量级别的范围，则要取决于纹理格式和每个像素的采样数量。

根据给定的纹理格式和采样数量，我们就能用`ID3D12Device::CheckFeatureSupport`[15]方法查询到对应的质量级别[16]：

```
typedef struct D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS {
 DXGI_FORMAT                Format;
 UINT                       SampleCount;
 D3D12_MULTISAMPLE_QUALITY_LEVEL_FLAGS Flags;
 UINT                       NumQualityLevels;
} D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS;

D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS msQualityLevels;
msQualityLevels.Format = mBackBufferFormat;
msQualityLevels.SampleCount = 4;
msQualityLevels.Flags = D3D12_MULTISAMPLE_QUALITY_LEVELS_FLAG_NONE;
msQualityLevels.NumQualityLevels = 0;
ThrowIfFailed(md3dDevice->CheckFeatureSupport(
  D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS,
  &msQualityLevels,
  sizeof(msQualityLevels)));
```

注意，此方法的第二个参数兼具输入和输出的属性。当它作为输入参数时，我们必须指定纹理格式、采样数量以及希望查询的多重采样所支持的标志（即立flag，或作旗标）。接着，待函数执行后便会填写图像质量级别作为输出。对于某种纹理格式和采样数量的组合来讲，其质量级别的有效范围为 0 至`NumQualityLevels–1`。

每个像素的最大采样数量被定义为：

```
#define   D3D12_MAX_MULTISAMPLE_SAMPLE_COUNT ( 32 )
```

但是，考虑到多重采样会占用内存资源，又为了保证程序性能等原因，通常会把采样数量设定为4或8。如果不希望使用多重采样，则可将采样数量设置为1，并令质量级别为0。其实在所有支持Direct3D 11的设备上，就已经可以对所有的渲染目标格式采用4X多重采样了。

注意 Note

在创建交换链缓冲区和深度缓冲区时都需要填写DXGI_SAMPLE_DESC结构体。当创建后台缓冲区和深度缓冲区时，多重采样的有关设置一定要相同[17]。

4.1.9　功能级别

从Direct3D 11开始便引进了功能级别（feature level）的概念（在代码里用枚举类型D3D_FEATURE_LEVEL表示），以下参数大致对应于Direct3D 9到Direct3D 11之间的各种版本[18]：

```
enum D3D_FEATURE_LEVEL
{
  D3D_FEATURE_LEVEL_9_1              = 0x9100,
  D3D_FEATURE_LEVEL_9_2              = 0x9200,
  D3D_FEATURE_LEVEL_9_3              = 0x9300,
  D3D_FEATURE_LEVEL_10_0             = 0xa000,
  D3D_FEATURE_LEVEL_10_1             = 0xa100,
  D3D_FEATURE_LEVEL_11_0             = 0xb000,
  D3D_FEATURE_LEVEL_11_1             = 0xb100
}D3D_FEATURE_LEVEL;
```

“功能级别”为不同级别所支持的功能进行了严格的界定（每个功能级别所支持的特定功能可参见SDK文档）。例如，一款支持功能级别11的GPU，除了个别特例之外（像类似于多重采样数量这样的信息仍然需要查询，因为Direct3D规范允许这些Direct3D 11硬件在此方面有各自不同的实现），必须支持完整的Direct3D 11功能集。功能集使程序员的开发工作更加便捷——只要了解所支持的功能集，就能知道有哪些Direct3D功能可供使用。

如果用户的硬件不支持某特定功能级别，应用程序理当回退至版本更低的功能级别。例如，为了照顾更多用户，一款应用程序可能会支持Direct3D 11、10乃至9.3级别的硬件。应用程序当按照从最新到最旧的级别支持顺序展开检测：首先检测Direct3D 11是否被支持，其次检测Direct3D 10，最后检测Direct3D 9.3。在本书中，我们总是假设需要支持的功能级别为`D3D_FEATURE_LEVEL_11_0`。但是在现实的应用程序中，我们往往需要考虑支持稍旧的硬件，以获得更多的用户。

4.1.10 DirectX图形基础结构

DirectX图形基础结构（DirectX Graphics Infrastructure，DXGI，也有译作DirectX图形基础设施）是一种与Direct3D配合使用的API。设计DXGI的基本理念是使多种图形API中所共有的底层任务能借助一组通用API来进行处理。例如，为了保证动画的流畅性，2D渲染与3D渲染两组API都要用到交换链和页面翻转功能，这里所用的交换链接口

`IDXGISwapChain`（详见4.1.4节）实际上就属于DXGI API。DXGI还用于处理一些其他常用的图形功能，如切换全屏模式（full-screen mode。另一种是窗口模式，windowed mode），枚举显示适配器、显示设备及其支持的显示模式（分辨率、刷新率等）等这类图形系统信息。除此之外，它还定义了Direct3D支持的各种表面格式信息（`DXGI_FORMAT`）。

我们刚刚简单地叙述了DXGI的概念，下面来介绍一些在Direct3D初始化时会用到的相关接口。`IDXGIFactory` 是DXGI中的关键接口之一，主要用于创建`IDXGISwapChain`接口以及枚举显示适配器。而显示适配器则真正实现了图形处理能力。通常来说，显示适配器（display adapter）是一种硬件设备（例如独立显卡），然而系统也可以用软件显示适配器来模拟硬件的图形处理功能。一个系统中可能会存在数个适配器（比如装有数块显卡）。适配器用接口`IDXGIAdapter`来表示。我们可以用下面的代码来枚举一个系统中的所有适配器：

```
void D3DApp::LogAdapters()
{
  UINT i = 0;
  IDXGIAdapter* adapter = nullptr;
  std::vector<IDXGIAdapter*> adapterList;
  while(mdxgiFactory->EnumAdapters(i, &adapter) != DXGI_ERROR_NOT_FOUND)
  {
    DXGI_ADAPTER_DESC desc;
    adapter->GetDesc(&desc);

    std::wstring text = L"***Adapter: ";
    text += desc.Description;
    text += L"\n";

    OutputDebugString(text.c_str());

    adapterList.push_back(adapter);

    ++i;
```

```
  }

  for(size_t i = 0; i < adapterList.size(); ++i)
  {
    LogAdapterOutputs(adapterList[i]);
    ReleaseCom(adapterList[i]);
  }
}
```

运行上述代码会输出类似于下面的信息：

```
***Adapter: NVIDIA GeForce GTX 760
***Adapter: Microsoft Basic Render Driver
```

“Microsoft Basic Render Driver（Microsoft基本呈现驱动程序）”是Windows 8及后续系统版本中包含的软件适配器。

另外，一个系统也可能装有数个显示设备。我们称每一台显示设备都是一个显示输出（display output，有的文档也作adapter output，适配器输出）实例，用**IDXGIOutput**接口来表示。每个适配器都与一组显示输出相关联。举个例子，考虑这样一个系统，该系统共有两块显卡和3台显示器，其中一块显卡与两台显示器相连，第三台显示器则与另一块显卡相连。在这种情况下，一块适配器与两个显示输出相关联，而另一块则仅有一个显示输出与之关联。通过以下代码，我们就可以枚举出与某块适配器关联的所有显示输出：

```
void D3DApp::LogAdapterOutputs(IDXGIAdapter* adapter)
{
  UINT i = 0;
  IDXGIOutput* output = nullptr;
  while(adapter->EnumOutputs(i, &output) != DXGI_ERROR_NOT_FOUND)
  {
    DXGI_OUTPUT_DESC desc;
    output->GetDesc(&desc);
```

```
    std::wstring text = L"***Output: ";
    text += desc.DeviceName;
    text += L"\n";
    OutputDebugString(text.c_str());

    LogOutputDisplayModes(output, DXGI_FORMAT_B8G8R8A8_UNORM);

    ReleaseCom(output);

    ++i;
  }
}
```

注意，官方文档中指出，在系统显卡驱动正常工作的情况下，“Microsoft Basic Render Driver”不会关联任何显示输出。

每种显示设备都有一系列它所支持的显示模式，可以用下列DXGI_MODE_DESC结构体中的数据成员来加以表示：

```
typedef struct DXGI_MODE_DESC
{
  UINT Width;                     // 分辨率宽度
  UINT Height;                    // 分辨率高度
  DXGI_RATIONAL RefreshRate;      // 刷新率，单位为赫兹Hz
  DXGI_FORMAT Format;             // 显示格式
  DXGI_MODE_SCANLINE_ORDER ScanlineOrdering; // 逐行扫描vs.隔行扫描
  DXGI_MODE_SCALING Scaling;      // 图像如何相对于屏幕进行拉伸
} DXGI_MODE_DESC;

typedef struct DXGI_RATIONAL
{
  UINT Numerator;
  UINT Denominator;
} DXGI_RATIONAL;

typedef enum DXGI_MODE_SCANLINE_ORDER
{
  DXGI_MODE_SCANLINE_ORDER_UNSPECIFIED    = 0,
  DXGI_MODE_SCANLINE_ORDER_PROGRESSIVE    = 1,
  DXGI_MODE_SCANLINE_ORDER_UPPER_FIELD_FIRST = 2,
  DXGI_MODE_SCANLINE_ORDER_LOWER_FIELD_FIRST = 3
} DXGI_MODE_SCANLINE_ORDER;[19]
```

```
typedef enum DXGI_MODE_SCALING
{
  DXGI_MODE_SCALING_UNSPECIFIED  = 0,
  DXGI_MODE_SCALING_CENTERED   = 1,        // 不做缩放，将图像显示在屏幕正中
  DXGI_MODE_SCALING_STRETCHED   = 2        // 根据屏幕的分辨率对图像进行拉伸
缩放
} DXGI_MODE_SCALING;
```

一旦确定了显示模式的具体格式（DXGI_FORMAT），我们就能通过下列代码，获得某个显示输出对此格式所支持的全部显示模式：

```
void D3DApp::LogOutputDisplayModes(IDXGIOutput* output, DXGI_FORMAT format
)
{
  UINT count = 0;
  UINT flags = 0;

  // 以nullptr作为参数调用此函数来获取符合条件的显示模式的个数
  output->GetDisplayModeList(format, flags, &count, nullptr);

  std::vector<DXGI_MODE_DESC> modeList(count);
  output->GetDisplayModeList(format, flags, &count, &modeList[0]);

  for(auto& x : modeList)
  {
    UINT n = x.RefreshRate.Numerator;
    UINT d = x.RefreshRate.Denominator;
    std::wstring text =
      L"Width = " + std::to_wstring(x.Width) + L" " +
      L"Height = " + std::to_wstring(x.Height) + L" " +
      L"Refresh = " + std::to_wstring(n) + L"/" + std::to_wstring(d) +
      L"\n";

    ::OutputDebugString(text.c_str());
  }
}
```

运行以上代码会输出下列相似的结果：

```
***Output: \\.\DISPLAY2
...
Width = 1920 Height = 1080 Refresh = 59950/1000
```

```
Width = 1920 Height = 1200 Refresh = 59950/1000
```

在进入全屏模式之时，枚举显示模式就显得尤为重要。为了获得最优的全屏性能，我们所指定的显示模式（包括刷新率）一定要与显示器支持的显示模式完全匹配。根据枚举出来的显示模式进行选定，便可以保证这一点。

有关DXGI的更多资料，可参阅“DXGI Overview”（DXGI概述）、“DirectX Graphics Infrastructure: Best Practices“（DirectX图形基础结构：最佳实践）以及“DXGI 1.4 Improvements”（DXGI 1.4的改良）等文章。

4.1.11 功能支持的检测

我们已经通过ID3D12Device::CheckFeatureSupport方法，检测了当前图形驱动对多重采样的支持。然而，这只是此函数对功能支持检测的冰山一角。这个方法的原型为：

```
HRESULT ID3D12Device::CheckFeatureSupport(
  D3D12_FEATURE Feature,
  void *pFeatureSupportData,
  UINT FeatureSupportDataSize);
```

1. Feature：枚举类型D3D12_FEATURE中的成员之一，用于指定我们希望检测的功能支持类型。

a）D3D12_FEATURE_D3D12_OPTIONS：检测当前图形驱动对Direct3D 12各种功能的支持情况。

b）D3D12_FEATURE_ARCHITECTURE：检测图形适配器中GPU的硬件体系架构特性。

c）D3D12_FEATURE_FEATURE_LEVELS：检测对功能级别的支持情况。

d）D3D12_FEATURE_FORMAT_SUPPORT：检测对给定纹理格式的支持情况（例如，指定的格式能否用于渲染目标？或，指定的格式能否用于混合技术？）。

e）D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS：检测对多重采样功能的支持情况。

2. pFeatureSupportData：指向某种数据结构的指针，该结构中存有检索到的特定功能支持的信息。此结构体的具体类型取决于Feature参数。

a）如果将Feature参数指定为D3D12_FEATURE_D3D12_OPTIONS，则传回的是一个D3D12_FEATURE_DATA_D3D12_OPTIONS实例。

b）如果将Feature参数指定为D3D12_FEATURE_ARCHITECTURE，则传回的是一个D3D12_FEATURE_DATA_ARCHITECTURE实例。

c）如果将Feature参数指定为D3D12_FEATURE_FEATURE_LEVELS，则传回的是一个D3D12_FEATURE_DATA_FEATURE_LEVELS实例。

d）如果将Feature参数指定为D3D12_FEATURE_FORMAT_SUPPORT，则传回的是一个D3D12_FEATURE_DATA_FORMAT_SUPPORT实例。

e）如果将Feature参数指定为D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS，则传回的是一个D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS实例。

3. FeatureSupportDataSize：传回pFeatureSupportData参数中的数据结构的大小。

ID3D12Device::CheckFeatureSupport函数能检测的支持功能有很多，但本书不会对那些高级的功能进行检测；至于每种功能结构体中数据成员的细节，可参见SDK文档。下面举一个例子，里面展示了如何对功能级别（详见4.1.9节）的支持情况进行检测：

```
typedef struct D3D12_FEATURE_DATA_FEATURE_LEVELS {
 UINT          NumFeatureLevels;
 const D3D_FEATURE_LEVEL *pFeatureLevelsRequested;
 D3D_FEATURE_LEVEL    MaxSupportedFeatureLevel;
} D3D12_FEATURE_DATA_FEATURE_LEVELS;

D3D_FEATURE_LEVEL featureLevels[3] =
{
  D3D_FEATURE_LEVEL_11_0,  // 首先检测是否支持D3D 11
  D3D_FEATURE_LEVEL_10_0,  // 其次检测是否支持D3D 10
  D3D_FEATURE_LEVEL_9_3    // 最后检测是否支持D3D 9.3
};

D3D12_FEATURE_DATA_FEATURE_LEVELS featureLevelsInfo;
featureLevelsInfo.NumFeatureLevels = 3;
featureLevelsInfo.pFeatureLevelsRequested = featureLevels;
md3dDevice->CheckFeatureSupport(
  D3D12_FEATURE_FEATURE_LEVELS,
```

```
    &featureLevelsInfo,
    sizeof(featureLevelsInfo));
```

注意，CheckFeatureSupport 方法的第二个参数兼有输入和输出的属性。作为输入的时候，先要指定功能级别数组中元素的个数（NumFeatureLevels），再令（pFeatureLevelsRequested）指针指向功能级别数组，其中应包括我们希望检测的一系列硬件支持功能级别。最后，此函数将用MaxSupportedFeatureLevel字段返回当前硬件可支持的最高功能级别。

4.1.12　资源驻留

复杂的游戏会运用大量纹理和3D网格（3d mesh，详见5.2节）等资源，但是其中的大多数并不需要总是置于显存[20]中供GPU使用。例如，让我们来构想这样一个游戏场景：在野外的森林中，有一个巨大的洞穴。在玩家进入洞穴之前，绘制画面并不会用到与洞穴相关的资源；当玩家进入洞穴之后，又不再需要森林数据资源。

在Direct3D 12中，应用程序通过控制资源在显存中的去留，主动管理资源的驻留情况（即residency。无论资源是否本已位于显存中，都可对其进行管理。在Direct3D 11中则由系统自动管理）。该技术的基本思路为使应用程序占用最小的显存空间。这是因为显存的空间有限，很可能不足以容下整个游戏的所有资源，或者用户还有运行中的程序也在同时使用显存。这里给出一条与性能相关的提示：程序应当避免在短时间内于显存中交换进出相同的资源，这会引起过高的开销。最理想的情况

是，所清出的资源在短时间内不会再次使用。游戏关卡或游戏场景的切换是关于常驻资源的好例子。

一般来说，资源在创建时就会驻留在显存中，而当它被销毁时则清出。但是通过下面的方法，我们就可以自己来控制资源的驻留：

```
HRESULT ID3D12Device::MakeResident(
  UINT          NumObjects,
  ID3D12Pageable *const *ppObjects);

HRESULT ID3D12Device::Evict(
  UINT          NumObjects,
  ID3D12Pageable *const *ppObjects);
```

这两种方法的第二个参数都是ID3D12Pageable资源数组，第一个参数则表示该数组中资源的数量。

为了简单起见，我们会把本书中演示程序的规模控制得比游戏小得多，所以也就不必对资源的驻留进行管理。对于资源驻留的更多信息可参见《Residency（驻留）》[21]。

4.2　CPU与GPU间的交互

在进行图形编程的时候，我们一定要了解有两种处理器在参与处理工作，即CPU和GPU，两者并行工作，但时而也需同步。为了获得最佳性能，最好的情况是让两者尽量同时工作，少同步。同步是一种我们不乐于执行的操作，因为这意味着一种处理器要以空闲状态等待另一种处理器完成某些任务，换句话说，它破坏了两者并行工作的机制。

4.2.1　命令队列和命令列表

每个GPU都至少维护着一个命令队列（command queue，本质上是环形缓冲区，即ring buffer）。借助Direct3D API，CPU可利用命令列表（command list）将命令提交到这个队列中去[22]（见图4.6）。当一系列命令被提交至命令队列之时，它们并不会被GPU立即执行，理解这一点至关重要。由于GPU可能正在处理先前插入命令队列内的命令，因此，后来新到的命令会一直在这个队列之中等待执行。

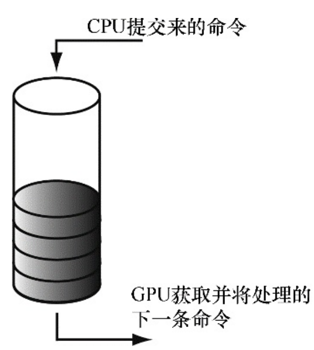


图4.6　命令队列

假如命令队列中变得空空如也，那么没有任务可执行的GPU只能空闲下来；相反地，如果命令队列被填满，那么CPU必将随着GPU的工作步伐在某些时刻保持空闲[Crawfis12]。这两种情况都是我们不希望碰到的。对于像游戏这样的高性能应用程序来说，它们的目标是充分利用硬件资源，保持CPU和GPU同时忙碌。

在Direct3D 12中，命令队列被抽象为ID3D12CommandQueue接口来表示。要通过填写D3D12_COMMAND_QUEUE_DESC结构体来描述队列，再调用ID3D12Device::CreateCommandQueue方法创建队列。我们在本书中将实际采用以下流程：

```
Microsoft::WRL::ComPtr<ID3D12CommandQueue> mCommandQueue;
D3D12_COMMAND_QUEUE_DESC queueDesc = {};
queueDesc.Type = D3D12_COMMAND_LIST_TYPE_DIRECT;
queueDesc.Flags = D3D12_COMMAND_QUEUE_FLAG_NONE;
ThrowIfFailed(md3dDevice->CreateCommandQueue(
  &queueDesc, IID_PPV_ARGS(&mCommandQueue)));
```

IID_PPV_ARGS辅助宏的定义如下：

```
#define IID_PPV_ARGS(ppType) __uuidof(**(ppType)), IID_PPV_ARGS_Helper(ppT
ype)
```

其中，__uuidof(**(ppType))将获取(**(ppType))的COM接口ID（globally unique identifier，全局唯一标识符，GUID），在上述代码段中得到的即为ID3D12CommandQueue接口的COM ID。IID_PPV_ARGS辅助函数的本质是将ppType强制转换为void**类型。我们在全书中都会见到此宏的身影，这是因为在调用Direct3D 12中创建接口实例的API时，大多都有一个参数是类型为void**的待创接口COM ID。

ExecuteCommandLists是一种常用的ID3D12CommandQueue接口方法，利用它可将命令列表里的命令添加到命令队列之中：

```
void ID3D12CommandQueue::ExecuteCommandLists(
  // 第二个参数里命令列表数组中命令列表的数量
  UINT Count,
  // 待执行的命令列表数组，指向命令列表数组中第一个元素的指针
  ID3D12CommandList *const *ppCommandLists);
```

GPU将从数组里的第一个命令列表开始顺序执行。

ID3D12GraphicsCommandList[23]接口封装了一系列图形渲染命令，它实际上继承于ID3D12CommandList接口。ID3D12GraphicsCommandList接口有数种方法向命令列表添加命令。例如，下面的代码依次就向命令列表中添加了设置视口、清除渲染目标视图和发起绘制调用的命令：

```
// mCommandList为一个指向ID3D12CommandList接口的指针
mCommandList->RSSetViewports(1, &mScreenViewport);
mCommandList->ClearRenderTargetView(mBackBufferView,
  Colors::LightSteelBlue, 0, nullptr);
mCommandList->DrawIndexedInstanced(36, 1, 0, 0, 0);
```

虽然这些方法的名字看起来像是会使对应的命令立即执行，但事实却并非如此，上面的代码仅仅是将命令加入命令列表而已。调用ExecuteCommandLists方法才会将命令真正地送入命令队列，供GPU在合适的时机处理。随着本书内容的不断深入，我们将逐步掌握ID3D12GraphicsCommandList所支持的各种命令。当命令都被加入命令列表之后，我们必须调用ID3D12GraphicsCommandList::Close方法来结束命令的记录：

```
// 结束记录命令
mCommandList->Close();
```

在调用ID3D12CommandQueue::ExecuteCommandLists方法提交命令列表之前，一定要将其关闭。

还有一种与命令列表有关的名为ID3D12CommandAllocator的内存管理类接口。记录在命令列表内的命令，实际上是存储在与之关联的命令分配器（command allocator）上。当通过ID3D12CommandQueue::ExecuteCommandLists方法执行命令列表的时候，命令队列就会引用分配器里的命令。而命令分配器则由ID3D12Device接口来创建：

```
HRESULT ID3D12Device::CreateCommandAllocator(
  D3D12_COMMAND_LIST_TYPE type,
  REFIID riid,
  void **ppCommandAllocator);
```

1．type：指定与此命令分配器相关联的命令列表类型。以下是本书常用的两种命令列表类型[24]。

a）D3D12_COMMAND_LIST_TYPE_DIRECT。存储的是一系列可供GPU直接执行的命令（这种类型的命令列表我们之前曾提到过）。

b）D3D12_COMMAND_LIST_TYPE_BUNDLE。将命令列表打包（bundle，也有译作集合）。构建命令列表时会产生一定的CPU开销，为此，Direct3D 12提供了一种优化的方法，允许我们将一系列命令打成所谓的包。当打包完成（命令记录完毕）之后，驱动就会对其中的命令

进行预处理，以使它们在渲染期间的执行过程中得到优化。因此，我们应当在初始化期间就用包记录命令。如果经过分析，发现构造某些命令列表会花费大量的时间，就可以考虑使用打包技术对其进行优化。Direct3D 12中的绘制API的效率很高，所以一般不会用到打包技术。因此，也许在证明其确实可以带来性能的显著提升时才会用到它。这就是说，在大多数情况下，我们往往会将其束之高阁。本书中不会使用打包技术，关于它的详情可参见DirectX 12文档[25]。

2. riid：待创建ID3D12CommandAllocator接口的COM ID。

3. ppCommandAllocator：输出指向所建命令分配器的指针。

命令列表同样由ID3D12Device接口创建：

```
HRESULT ID3D12Device::CreateCommandList(
  UINT nodeMask,
  D3D12_COMMAND_LIST_TYPE type,
  ID3D12CommandAllocator *pCommandAllocator,
  ID3D12PipelineState *pInitialState,
  REFIID riid,
  void **ppCommandList);
```

1. nodeMask：对于仅有一个GPU的系统而言，要将此值设为0；对于具有多GPU的系统而言，此节点掩码（node mask）指定的是与所建命令列表相关联的物理GPU。本书中假设我们使用的是单GPU系统。

2. type：命令列表的类型，常用的选项为D3D12_COMMAND_LIST_TYPE_DIRECT和D3D12_COMMAND_LIST_TYPE_BUNDLE。

3. `pCommandAllocator`：与所建命令列表相关联的命令分配器。它的类型必须与所创命令列表的类型相匹配。

4. `pInitialState`：指定命令列表的渲染流水线初始状态。对于打包技术来说可将此值设为`nullptr`，另外，此法同样适用于执行命令列表中不含有任何绘制命令，即执行命令列表是为了达到初始化的目的的特殊情况。我们将在第6章中详细讨论`ID3D12PipelineState`接口。

5. `riid`：待创建`ID3D12CommandList`接口的COM ID。

6. `ppCommandList`：输出指向所建命令列表的指针。

注意 Note

我们可以通过`ID3D12Device::GetNodeCount`方法来查询系统中GPU适配器节点（物理GPU）的数量。

我们可以创建出多个关联于同一命令分配器的命令列表，但是不能同时用它们来记录命令。因此，当其中的一个命令列表在记录命令时，必须关闭同一命令分配器的其他命令列表。换句话说，要保证命令列表中的所有命令都会按顺序连续地添加到命令分配器内。还要注意的一点是，当创建或重置一个命令列表的时候，它会处于一种“打开”的状态。所以，当尝试为同一个命令分配器连续创建两个命令列表时，我们会得

到这样的一个错误消息：

```
D3D12 ERROR: ID3D12CommandList::{Create,Reset}CommandList: The command all
ocator is currently in-use by another command list.
（D3D12 错误: ID3D12CommandList::{Create,Reset}CommandList:此命令分配器正在
被另一个命令列表占用）
```

在调用 ID3D12CommandQueue::ExecuteCommandList(C) 方法之后，我们就可以通过ID3D12GraphicsCommandList::Reset方法，安全地复用命令列表C占用的相关底层内存来记录新的命令集。Reset方法中的参数对应于以ID3D12Device::CreateCommandList方法创建命令列表时所用的参数。

```
HRESULT ID3D12GraphicsCommandList::Reset(
  ID3D12CommandAllocator *pAllocator,
  ID3D12PipelineState *pInitialState);
```

此方法将命令列表恢复为刚创建时的初始状态，我们可以借此继续复用其低层内存，也可以避免释放旧列表再创建新列表这一系列的烦琐操作。注意，重置命令列表并不会影响命令队列中的命令，因为相关的命令分配器仍在维护着其内存中被命令队列引用的系列命令。

向GPU提交了一整帧的渲染命令后，我们可能还要为了绘制下一帧而复用命令分配器中的内存。ID3D12CommandAllocator::Reset方法由此应运而生：

```
HRESULT ID3D12CommandAllocator::Reset(void);
```

这种方法的功能类似于向量类中的std::vector::clear方法，后者使向量的大小（size）归零，但是仍保持其当前的容量（capacity）。

然而，由于命令队列可能会引用命令分配器中的数据，所以在没有确定**GPU**执行完命令分配器中的所有命令之前，千万不要重置命令分配器！下一节将介绍相关内容。

4.2.2　CPU与GPU间的同步

当两种处理器并行工作时，自然而然地就会产生一系列的同步问题。

假设有一资源R，里面存有待绘制几何体的位置信息。现在，令CPU对R中的数据进行更新，先把R中的几何体位置信息改为p_1，再向命令队列里添加绘制资源R的命令C，以此将几何体绘制到位置p_1。由于向命令队列添加命令并不会阻塞CPU，所以CPU会继续执行后序指令。在GPU执行绘制命令C之前，如果CPU率先覆写了数据R，提前把其中的位置信息修改为p_2，那么这个行为就会造成一个严重的错误（见图4.7）。

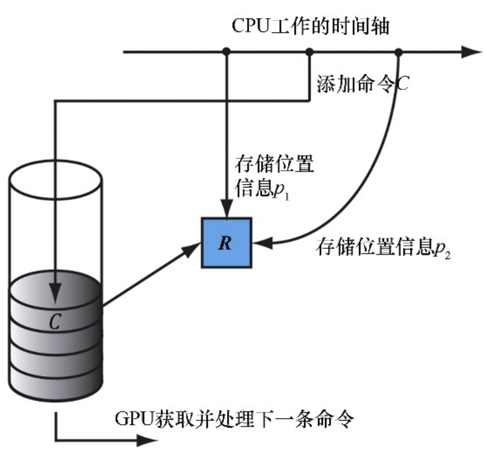


图4.7　不管是命令C按坐标p_2绘制几何体，

还是在绘制的过程中更新资源R都是错误的
行为。这两种情况都不是我们所预期的

解决此问题的一种办法是：强制CPU等待，直到GPU完成所有命令的处理，达到某个指定的围栏点（fence point）为止。我们将这种方法称为刷新命令队列（flushing the command queue），可以通过围栏（fence）来实现这一点。围栏用`ID3D12Fence`接口来表示[26]，此技术能用于实现GPU和CPU间的同步。创建一个围栏对象的方法如下：

```
HRESULT ID3D12Device::CreateFence(
  UINT64 InitialValue,
  D3D12_FENCE_FLAGS Flags,
  REFIID riid,
  void **ppFence);

// 示例
ThrowIfFailed(md3dDevice->CreateFence(
  0,
  D3D12_FENCE_FLAG_NONE,
  IID_PPV_ARGS(&mFence)));
```

每个围栏对象都维护着一个`UINT64`类型的值，此为用来标识围栏点的整数。起初，我们将此值设为0，每当需要标记一个新的围栏点时就将它加1。现在，我们用代码和注释进行展示，看看如何用一个围栏来刷新命令队列。

```
UINT64 mCurrentFence = 0;
void D3DApp::FlushCommandQueue()
{
  // 增加围栏值，接下来将命令标记到此围栏点
  mCurrentFence++;

  // 向命令队列中添加一条用来设置新围栏点的命令
  // 由于这条命令要交由GPU处理（即由GPU端来修改围栏值），所以在GPU处理完命令队列中此Signal()
  // 以前的所有命令之前，它并不会设置新的围栏点[27]
```

```
  ThrowIfFailed(mCommandQueue->Signal(mFence.Get(), mCurrentFence));

  // 在CPU端等待GPU，直到后者执行完这个围栏点之前的所有命令
  if(mFence->GetCompletedValue() < mCurrentFence)
  {
    HANDLE eventHandle = CreateEventEx(nullptr, false, false, EVENT_ALL_AC
CESS);

    // 若GPU命中当前的围栏（即执行到Signal()指令，修改了围栏值），则激发预定事
件
    ThrowIfFailed(mFence->SetEventOnCompletion(mCurrentFence, eventHandle)
);

    // 等待GPU命中围栏，激发事件
    WaitForSingleObject(eventHandle, INFINITE);
    CloseHandle(eventHandle);
  }
}
```

图4.8为这段代码的时间轴示意图。

这样一来，在本节开始给出的情景中，当CPU发出绘制命令C后，在将R内的位置信息改写为p_2之前，应率先刷新命令队列。这种解决方案其实并不完美，因为这意味着在等待GPU处理命令的时候，CPU会处于空闲状态，但在第7章以前也只能暂时使用这个简单的办法了。我们几乎可以在任何时间点上刷新命令队列（当然，不一定仅在渲染每一帧时才刷新一次）。例如，若有一些GPU初始化命令有待执行，我们便可以在进入渲染主循环之前刷新命令队列，从而进行这些初始化操作。

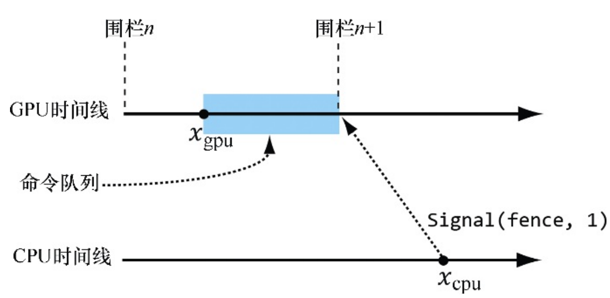


图4.8　GPU已经执行到了命令x_{gpu}，而CPU则刚刚在x_{cpu}处调用了

`ID3D12CommandQueue::Signal(fence, n+1)`方法让GPU端设置围栏值。该方法实际上是在命令队列的结尾处添加一条命令，使围栏值变为$n+1$$n$的初始值为0）。而在GPU处理完命令队列中`Signal(fence, n+1)`之前的所有命令以前，CPU端调用的`mFence->GetCompletedValue()`方法会一直返回值n

其实，用刷新命令队列的办法也可以解决上一小节末尾遇到的问题，即在重置命令分配器之前，先刷新命令队列来确定GPU的命令都已执行完毕。

4.2.3　资源转换

为了实现常见的渲染效果，我们经常会通过GPU对某个资源R按顺序进行先写后读这两种操作。然而，当GPU的写操作还没有完成抑或甚至还没有开始，却开始读取资源，便会导致资源冒险（resource hazard）。为此，Direct3D专门针对资源设计了一组相关状态。资源在创建伊始会处于默认状态，该状态将一直持续到应用程序通过Direct3D将其转换（transition）为另一种状态为止。这就使GPU能够针对资源状态转换与防止资源冒险作出适当的行为。例如，如果要对某个资源（比如纹理）执行写操作时，需要将它的状态转换为渲染目标状态；而要对该纹理进行读操作时，再把它的状态变为着色器资源状态。根据Direct3D给出的转换信息，GPU就可以采取适当的措施避免资源冒险的发生。譬如，在读取某个资源之前，它会等待所有与之相关的写操作执行完毕。应用程序开发者应当知道，资源转换所带来的负荷会造成程序性能的下降。除此之外，一个自动跟踪状态转换的系统也在强行增加程序的额外开销[28]。

通过命令列表设置转换资源屏障（transition resource barrier）数组，即可指定资源的转换；当我们希望以一次API调用来转换多个资源的时候，这种数组就派上了用场。在代码中，资源屏障用D3D12_RESOURCE_BARRIER结构体来表示[29]。下列辅助函数（定义于d3dx12.h头文件之中）将根据用户给出的资源和指定的前后转换状态，返回对应的转换资源屏障描述：

```
struct CD3DX12_RESOURCE_BARRIER : public D3D12_RESOURCE_BARRIER
{
  // [...] 辅助方法

static inline CD3DX12_RESOURCE_BARRIER Transition(
    _In_ ID3D12Resource* pResource,
    D3D12_RESOURCE_STATES stateBefore,
    D3D12_RESOURCE_STATES stateAfter,
    UINT subresource = D3D12_RESOURCE_BARRIER_ALL_SUBRESOURCES,
    D3D12_RESOURCE_BARRIER_FLAGS flags = D3D12_RESOURCE_BARRIER_FLAG_NONE)
  {
    CD3DX12_RESOURCE_BARRIER result;
    ZeroMemory(&result, sizeof(result));
    D3D12_RESOURCE_BARRIER &barrier = result;
    result.Type = D3D12_RESOURCE_BARRIER_TYPE_TRANSITION;
    result.Flags = flags;
    barrier.Transition.pResource = pResource;
    barrier.Transition.StateBefore = stateBefore;
    barrier.Transition.StateAfter = stateAfter;
    barrier.Transition.Subresource = subresource;
    return result;
  }
  // [...] 其他辅助方法
};
```

可以看到，CD3DX12_RESOURCE_BARRIER继承自D3D12_RESOURCE_BARRIER结构体，并添加了一些辅助方法。Direct3D 12中的许多结构体都有其对应的扩展辅助结构变体（variation），考虑到使用上的方便性，我们更偏爱于运用那些变体。以CD3DX12作为前缀

的变体全都定义在d3dx12.h头文件当中，这个文件并不属于DirectX 12 SDK的核心部分，但是可以通过微软的官方网站下载获得。为了方便起见，本书源代码的Common目录里附有一份d3dx12.h头文件。

在本章的示例程序中，此辅助函数的用法如下：

```
mCommandList->ResourceBarrier(1,
  &CD3DX12_RESOURCE_BARRIER::Transition(
    CurrentBackBuffer(),
    D3D12_RESOURCE_STATE_PRESENT,
    D3D12_RESOURCE_STATE_RENDER_TARGET));
```

这段代码将以图片形式显示在屏幕中的纹理，从呈现状态转换为渲染目标状态。那么，这个添加到命令列表中的资源屏障究竟是何物呢？事实上，我们可以将此资源屏障转换看作是一条告知GPU某资源状态正在进行转换的命令。所以在执行后续的命令时，GPU便会采取必要措施以防资源冒险。

注意 Note

Direct3D 12提供的转换类型不止文中提到寥寥几种。但是，我们暂时只会用到上述转换屏障。至于其他类型的屏障，我们将随用随讲。

4.2.4　命令与多线程

Direct3D 12的设计目标是为用户提供一个高效的多线程环境，命令列表也是一种发挥Direct3D多线程优势的途径。对于内含许多物体的庞大场景而言，仅通过一个构建命令列表来绘制整个场景会占用不少的CPU时间。因此，可以采取一种并行创建命令列表的思路。例如，我们可以创建4条线程，每条分别负责构建一个命令列表来绘制25%的场景物体。

以下是一些在多线程环境中使用命令列表要注意的问题。

1．命令列表并非自由线程（not free-threaded）对象。也就是说，多线程既不能同时共享相同的命令列表，也不能同时调用同一命令列表的方法。所以，每个线程通常都只使用各自的命令列表。

2．命令分配器亦不是线程自由的对象。这就是说，多线程既不能同时共享同一个命令分配器，也不能同时调用同一命令分配器的方法。所以，每个线程一般都仅使用属于自己的命令分配器。

3．命令队列是线程自由对象，所以多线程可以同时访问同一命令队列，也能够同时调用它的方法。特别是每个线程都能同时向命令队列提交它们自己所生成的命令列表。

4．出于性能的原因，应用程序必须在初始化期间，指出用于并行记录命令的命令列表最大数量。

为了简单起见，本书不会使用多线程技术。完成本书的阅读后，读者可以通过查阅SDK中的Multithreading12示例[30]来学习怎样并行生成

命令列表。如果希望应用程序充分利用系统资源，应该通过多线程技术来发挥CPU多核心的并行处理能力。

4.3　初始化Direct3D

这一节我们会利用自己编写的演示框架来展示Direct3D的初始化过程。这是一个比较冗长的流程，但每个程序只需执行一次即可。我们对Direct3D进行初始化的过程可以分为以下几个步骤：

1. 用`D3D12CreateDevice`函数创建`ID3D12Device`接口实例。
2. 创建一个`ID3D12Fence`对象，并查询描述符的大小。
3. 检测用户设备对4X MSAA质量级别的支持情况。
4. 依次创建命令队列、命令列表分配器和主命令列表。
5. 描述并创建交换链。
6. 创建应用程序所需的描述符堆。
7. 调整后台缓冲区的大小，并为它创建渲染目标视图。
8. 创建深度/模板缓冲区及与之关联的深度/模板视图。
9. 设置视口（viewport）和裁剪矩形（scissor rectangle）。

4.3.1　创建设备

要初始化Direct3D，必须先创建Direct3D 12设备

（ID3D12Device）。此设备代表着一个显示适配器。一般来说，显示适配器是一种3D图形硬件（如显卡）。但是，一个系统也能用软件显示适配器来模拟3D图形硬件的功能（如WARP适配器）。Direct3D 12设备既可检测系统环境对功能的支持情况，又能创建所有其他的Direct3D接口对象（如资源、视图和命令列表）。通过下面的函数就可以创建Direct3D 12设备：

```
HRESULT WINAPI D3D12CreateDevice(
  IUnknown* pAdapter,
  D3D_FEATURE_LEVEL MinimumFeatureLevel,
  REFIID riid, // ID3D12Device的COM ID
  void** ppDevice );
```

1．pAdapter：指定在创建设备时所用的显示适配器。若将此参数设定为空指针，则使用主显示适配器。我们在本书的示例中总是采用主适配器。在4.1.10节中，我们已展示了怎样枚举系统中所有的显示适配器。

2．MinimumFeatureLevel：应用程序需要硬件所支持的最低功能级别。如果适配器不支持此功能级别，则设备创建失败。在我们的框架中指定的是D3D_FEATURE_LEVEL_11_0（即支持Direct3D 11的特性）。

3．riid：所建ID3D12Device接口的COM ID。

4．ppDevice：返回所创建的Direct3D 12设备。

以下是此函数的调用示例：

```
#if defined(DEBUG) || defined(_DEBUG)
```

```
// 启用D3D12的调试层
{
  ComPtr<ID3D12Debug> debugController;
  ThrowIfFailed(D3D12GetDebugInterface(IID_PPV_ARGS(&debugController)));
  debugController->EnableDebugLayer();
}
#endif

ThrowIfFailed(CreateDXGIFactory1(IID_PPV_ARGS(&mdxgiFactory)));

// 尝试创建硬件设备
HRESULT hardwareResult = D3D12CreateDevice(
  nullptr,       // 默认适配器
  D3D_FEATURE_LEVEL_11_0,
  IID_PPV_ARGS(&md3dDevice));

// 回退至WARP设备
if(FAILED(hardwareResult))
{
  ComPtr<IDXGIAdapter> pWarpAdapter;
  ThrowIfFailed(mdxgiFactory->EnumWarpAdapter(IID_PPV_ARGS(&pWarpAdapter))
);

  ThrowIfFailed(D3D12CreateDevice(
    pWarpAdapter.Get(),
    D3D_FEATURE_LEVEL_11_0,
    IID_PPV_ARGS(&md3dDevice)));
}
```

可以看到，为了进入调试模式，我们首先开启了调试层（debug layer）。随后，Direct3D便会开启额外的调试功能，并在错误发生时向VC++的输出窗口发送类似于下面的调试信息：

```
D3D12 ERROR: ID3D12CommandList::Reset: Reset fails because the command lis
t was not closed.
（D3D12 ERROR: ID3D12CommandList::Reset: 由于没有关闭命令列表因此重置失败。）
```

还可以发现，当调用`D3D12CreateDevice`失败后，程序将回退到一种软件适配器：WARP设备。WARP意为Windows Advanced Rasterization Platform（Windows高级光栅化平台）。在Windows 7及以

下版本的操作系统中，WARP设备支持的最高功能级别是10.1；在Windows 8系统中，WARP设备支持的最高功能级别是11.1[31]。为了创建WARP适配器，需要先创建一个`IDXGIFactory4`对象，并通过它来枚举WARP适配器：

```
ComPtr<IDXGIFactory4> mdxgiFactory;
CreateDXGIFactory1(IID_PPV_ARGS(&mdxgiFactory));
mdxgiFactory->EnumWarpAdapter(
  IID_PPV_ARGS(&pWarpAdapter));
```

作为DXGI的一部分，`mdxgiFactory`对象也可用于创建交换链。

4.3.2　创建围栏并获取描述符的大小

一旦创建好设备，便可以为CPU/GPU的同步而创建围栏了。另外，若用描述符进行工作，还需要了解它们的大小。但描述符在不同的GPU平台上大小各异，这就需要我们去查询相关的信息。随后，我们会把描述符的大小缓存起来，需要时即可直接引用：

```
ThrowIfFailed(md3dDevice->CreateFence(
  0, D3D12_FENCE_FLAG_NONE, IID_PPV_ARGS(&mFence)));
mRtvDescriptorSize = md3dDevice->GetDescriptorHandleIncrementSize(
  D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
mDsvDescriptorSize = md3dDevice->GetDescriptorHandleIncrementSize(
  D3D12_DESCRIPTOR_HEAP_TYPE_DSV);
mCbvUavDescriptorSize = md3dDevice->GetDescriptorHandleIncrementSize(
  D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
```

4.3.3　检测对4X MSAA质量级别的支持

在本书中，我们要对4X MSAA的支持情况进行检测。这里选择4X，是因为借此采样数量就可以获得开销不高却性能不凡的效果。而且，在一切支持Direct3D 11的设备上，所有的渲染目标格式就皆已支持4X MSAA了。因此，凡是支持Direct3D 11的硬件，都会保证此项功能的正常开启，我们也就无须再对此进行检验了。但是，对质量级别的检测还是不可或缺，为此，可采取下列方法加以实现：

```
D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS msQualityLevels;
msQualityLevels.Format = mBackBufferFormat;
msQualityLevels.SampleCount = 4;
msQualityLevels.Flags = D3D12_MULTISAMPLE_QUALITY_LEVELS_FLAG_NONE;
msQualityLevels.NumQualityLevels = 0;
ThrowIfFailed(md3dDevice->CheckFeatureSupport(
  D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS,
  &msQualityLevels,
  sizeof(msQualityLevels)));

m4xMsaaQuality = msQualityLevels.NumQualityLevels;
assert(m4xMsaaQuality > 0 && "Unexpected MSAA quality level.");
```

由于我们所用的平台必能支持4X MSAA这一功能，其返回值应该也总是大于0，所以对此而做出上述断言。

4.3.4 创建命令队列和命令列表

回顾4.2.1节可知：`ID3D12CommandQueue`接口表示命令队列，`ID3D12CommandAllocator`接口代表命令分配器，`ID3D12GraphicsCommandList`接口表示命令列表。据此，我们通过以下代码分别展示这几种对象的创建流程：

```
ComPtr<ID3D12CommandQueue> mCommandQueue;
```

```
ComPtr<ID3D12CommandAllocator> mDirectCmdListAlloc;
ComPtr<ID3D12GraphicsCommandList> mCommandList;
void D3DApp::CreateCommandObjects()
{
  D3D12_COMMAND_QUEUE_DESC queueDesc = {};
  queueDesc.Type = D3D12_COMMAND_LIST_TYPE_DIRECT;
  queueDesc.Flags = D3D12_COMMAND_QUEUE_FLAG_NONE;
  ThrowIfFailed(md3dDevice->CreateCommandQueue(
    &queueDesc, IID_PPV_ARGS(&mCommandQueue)));
  ThrowIfFailed(md3dDevice->CreateCommandAllocator(
    D3D12_COMMAND_LIST_TYPE_DIRECT,
    IID_PPV_ARGS(mDirectCmdListAlloc.GetAddressOf())));

    ThrowIfFailed(md3dDevice->CreateCommandList(
    0,
    D3D12_COMMAND_LIST_TYPE_DIRECT,
    mDirectCmdListAlloc.Get(), // 关联命令分配器
    nullptr,                   // 初始化流水线状态对象
    IID_PPV_ARGS(mCommandList.GetAddressOf())));

  // 首先要将命令列表置于关闭状态。这是因为在第一次引用命令列表时，我们要对它进行
重置，而在调用
  // 重置方法之前又需先将其关闭
  mCommandList->Close();
}
```

观察`CreateCommandList`方法会发现，我们将流水线状态对象（pipeline state object）这一参数指定为了空指针。在本章的示例程序中，由于我们不会发起任何绘制命令，所以也就不会用到流水线状态对象。在第6章中，我们会对流水线状态对象开展更为详细的讨论。

4.3.5 描述并创建交换链

初始化流程的下一步是创建交换链。首先，要填写一份`DXGI_SWAP_CHAIN_DESC`结构体实例，用它来描述欲创建交换链的特性。此结构体的定义如下：

```
typedef struct DXGI_SWAP_CHAIN_DESC
{
  DXGI_MODE_DESC BufferDesc;
  DXGI_SAMPLE_DESC SampleDesc;
  DXGI_USAGE BufferUsage;
  UINT BufferCount;
  HWND OutputWindow;
  BOOL Windowed;
  DXGI_SWAP_EFFECT SwapEffect;
  UINT Flags;
} DXGI_SWAP_CHAIN_DESC;
```

其中的DXGI_MODE_DESC类型则是另一种结构体，它的定义为：

```
typedef struct DXGI_MODE_DESC
{
  UINT Width;                                          // 缓冲区分辨率的宽度
  UINT Height;                                         // 缓冲区分辨率的高度
  DXGI_RATIONAL RefreshRate;
  DXGI_FORMAT Format;                                // 缓冲区的显示格式
  DXGI_MODE_SCANLINE_ORDER ScanlineOrdering;  // 逐行扫描 vs. 隔行扫描
  DXGI_MODE_SCALING Scaling;                         // 图像如何相对于屏幕进行拉
伸
} DXGI_MODE_DESC;
```

在下列数据成员的描述中，只涉及对于初学者来讲最为重要的常用标志和选项。至于其他标志和选项的描述，可参见SDK的相关文档。

1. BufferDesc：这个结构体描述了待创建后台缓冲区的属性。在这里我们仅关注它的宽度、高度和像素格式属性。至于其他成员的细节可查看SDK文档。

2. SampleDesc：多重采样的质量级别以及对每个像素的采样次数，可参见4.1.8节。对于单次采样来说，我们要将采样数量指定为1，质量级别指定为0。

3. BufferUsage：由于我们要将数据渲染至后台缓冲区（即用它作为渲染目标），因此将此参数指定为DXGI_USAGE_RENDER_TARGET_OUTPUT。

4. BufferCount：交换链中所用的缓冲区数量。我们将它指定为2，即采用双缓冲。

5. OutputWindow：渲染窗口的句柄。

6. Windowed：若指定为true，程序将在窗口模式下运行；如果指定为false，则采用全屏模式。

7. SwapEffect：指定为DXGI_SWAP_EFFECT_FLIP_DISCARD。

8. Flags：可选标志。如果将其指定为DXGI_SWAP_CHAIN_FLAG_ALLOW_MODE_SWITCH，那么，当程序切换为全屏模式时，它将选择最适于当前应用程序窗口尺寸的显示模式。如果没有指定该标志，当程序切换为全屏模式时，将采用当前桌面的显示模式。

描述完交换链之后，我们用IDXGIFactory::CreateSwapChain方法来创建它：

```
HRESULT IDXGIFactory::CreateSwapChain(
 IUnknown *pDevice,              // 指向ID3D12CommandQueue接口的指针
 DXGI_SWAP_CHAIN_DESC *pDesc,   // 指向描述交换链的结构体的指针
 IDXGISwapChain **ppSwapChain); // 返回所创建的交换链接口
```

下面的代码展示了如何通过本书的演示框架来方便地创建交换链。研究此函数的代码就会发现，我们是按照可以对它进行多次调用来设计的。即，在创建新的交换链之前，先要销毁旧的交换链。这样一来，我们就可以用不同的设置来重新创建交换链，借此在运行时修改多重采样的配置[32]。

```
DXGI_FORMAT mBackBufferFormat = DXGI_FORMAT_R8G8B8A8_UNORM;

void D3DApp::CreateSwapChain()
{
  // 释放之前所创的交换链，随后再进行重建
  mSwapChain.Reset();

  DXGI_SWAP_CHAIN_DESC sd;
  sd.BufferDesc.Width = mClientWidth;
  sd.BufferDesc.Height = mClientHeight;
  sd.BufferDesc.RefreshRate.Numerator = 60;
  sd.BufferDesc.RefreshRate.Denominator = 1;
  sd.BufferDesc.Format = mBackBufferFormat;
  sd.BufferDesc.ScanlineOrdering = DXGI_MODE_SCANLINE_ORDER_UNSPECIFIED;
  sd.BufferDesc.Scaling = DXGI_MODE_SCALING_UNSPECIFIED;
  sd.SampleDesc.Count = m4xMsaaState ? 4 : 1;
  sd.SampleDesc.Quality = m4xMsaaState ? (m4xMsaaQuality - 1) : 0;
  sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
  sd.BufferCount = SwapChainBufferCount;
  sd.OutputWindow = mhMainWnd;
  sd.Windowed = true;
  sd.SwapEffect = DXGI_SWAP_EFFECT_FLIP_DISCARD;
  sd.Flags = DXGI_SWAP_CHAIN_FLAG_ALLOW_MODE_SWITCH;
  //注意，交换链需要通过命令队列对其进行刷新
  ThrowIfFailed(mdxgiFactory->CreateSwapChain(
    mCommandQueue.Get(),
    &sd,
    mSwapChain.GetAddressOf()));
}
```

4.3.6　创建描述符堆

我们需要通过创建描述符堆来存储程序中要用到的描述符/视图（参见4.1.6节）。对此，Direct3D 12 以ID3D12DescriptorHeap接口表示描述符堆，并用ID3D12Device::CreateDescriptorHeap方法来创建它。在本章的示例程序中，我们将为交换链中SwapChainBufferCount个用于渲染数据的缓冲区资源创建对应的渲染目标视图（Render Target View，RTV），并为用于深度测试（depth test）的深度/模板缓冲区资源创建一个深度/模板视图（Depth/Stencil View，DSV）。所以，我们此时需要创建两个描述符堆，其一用来存储SwapChainBufferCount个RTV，而那另一个描述堆则用来存储那1个DSV。现通过下述代码来创建这两个描述符堆：

```
ComPtr<ID3D12DescriptorHeap> mRtvHeap;
ComPtr<ID3D12DescriptorHeap> mDsvHeap;
void D3DApp::CreateRtvAndDsvDescriptorHeaps()
{
  D3D12_DESCRIPTOR_HEAP_DESC rtvHeapDesc;
  rtvHeapDesc.NumDescriptors = SwapChainBufferCount;
  rtvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
  rtvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
    rtvHeapDesc.NodeMask = 0;
   ThrowIfFailed(md3dDevice->CreateDescriptorHeap(
    &rtvHeapDesc, IID_PPV_ARGS(mRtvHeap.GetAddressOf())));

  D3D12_DESCRIPTOR_HEAP_DESC dsvHeapDesc;
  dsvHeapDesc.NumDescriptors = 1;
  dsvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_DSV;
  dsvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
    dsvHeapDesc.NodeMask = 0;
  ThrowIfFailed(md3dDevice->CreateDescriptorHeap(
    &dsvHeapDesc, IID_PPV_ARGS(mDsvHeap.GetAddressOf())));
}
```

在本书的应用框架中有以下定义：

```
static const int SwapChainBufferCount = 2;
```

```
int mCurrBackBuffer = 0;
```

其中，mCurrBackBuffer是用来记录当前后台缓冲区的索引（由于利用页面翻转技术来交换前台缓冲区和后台缓冲区，所以我们需要对其进行记录，以便搞清楚哪个缓冲区才是当前正在用于渲染数据的后台缓冲区）。

创建描述符堆之后，还要能访问其中所存的描述符。在程序中，我们是通过句柄来引用描述符的，并以ID3D12DescriptorHeap::GetCPUDescriptorHandleForHeapStart方法来获得描述符堆中第一个描述符的句柄。借助下列函数即可获取当前后台缓冲区的RTV与DSV：

```
D3D12_CPU_DESCRIPTOR_HANDLE D3DApp::CurrentBackBufferView()const
{
  // CD3DX12构造函数根据给定的偏移量找到当前后台缓冲区的RTV
  return CD3DX12_CPU_DESCRIPTOR_HANDLE(
    mRtvHeap->GetCPUDescriptorHandleForHeapStart(),// 堆中的首个句柄
    mCurrBackBuffer,    // 偏移至后台缓冲区描述符句柄的索引
    mRtvDescriptorSize); // 描述符所占字节的大小
}

D3D12_CPU_DESCRIPTOR_HANDLE D3DApp::DepthStencilView()const
{
  return mDsvHeap->GetCPUDescriptorHandleForHeapStart();
}
```

通过这段示例代码，我们就能够看出描述符大小的用途了。为了用偏移量找到当前后台缓冲区的RTV描述符[33]，我们就必须知道RTV描述符的大小。

4.3.7 创建渲染目标视图

如4.1.6节中所述，资源不能与渲染流水线中的阶段直接绑定，所以我们必须先为资源创建视图（描述符），并将其绑定到流水线阶段。例如，为了将后台缓冲区绑定到流水线的输出合并阶段（output merger stage，这样Direct3D才能向其渲染），便需要为该后台缓冲区创建一个渲染目标视图。而这第一个步骤就是要获得存于交换链中的缓冲区资源。

```
HRESULT IDXGISwapChain::GetBuffer(
  UINT Buffer,
  REFIID riid,
  void **ppSurface);
```

1. `Buffer`：希望获得的特定后台缓冲区的索引（有时后台缓冲区并不只一个，所以需要用索引来指明）。

2. `riid`：希望获得的`ID3D12Resource`接口[34]的COM ID。

3. `ppSurface`：返回一个指向`ID3D12Resource`接口的指针，这便是希望获得的后台缓冲区。

调用`IDXGISwapChain::GetBuffer`方法会增加相关后台缓冲区的COM引用计数，所以在每次使用后一定要将其释放。通过`ComPtr`便可以自动做到这一点。

接下来，使用`ID3D12Device::CreateRenderTargetView`方法来为获取的后台缓冲区创建渲染目标视图。

```
void ID3D12Device::CreateRenderTargetView(
  ID3D12Resource *pResource,
```

```
  const D3D12_RENDER_TARGET_VIEW_DESC *pDesc,
  D3D12_CPU_DESCRIPTOR_HANDLE DestDescriptor);
```

1. pResource：指定用作渲染目标的资源。在上面的例子中是后台缓冲区（即为后台缓冲区创建了一个渲染目标视图）。

2. pDesc：指向D3D12_RENDER_TARGET_VIEW_DESC数据结构实例的指针。该结构体描述了资源中元素的数据类型（格式）。如果该资源在创建时已指定了具体格式（即此资源不是无类型格式，not typeless），那么就可以把这个参数设为空指针，表示采用该资源创建时的格式，为它的第一个mipmap层级（后台缓冲区只有一种mipmap层级，有关mipmap的内容将在第9章展开讨论）创建一个视图。由于已经指定了后台缓冲区的格式，因此就将这个参数设置为空指针。

3. DestDescriptor：引用所创建渲染目标视图的描述符句柄。

下面的示例通过调用这两种方法为交换链中的每一个缓冲区都创建了一个RTV：

```
ComPtr<ID3D12Resource> mSwapChainBuffer[SwapChainBufferCount];
CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHeapHandle(
  mRtvHeap->GetCPUDescriptorHandleForHeapStart());
for (UINT i = 0; i < SwapChainBufferCount; i++)
{
  // 获得交换链内的第i个缓冲区
  ThrowIfFailed(mSwapChain->GetBuffer(
    i, IID_PPV_ARGS(&mSwapChainBuffer[i])));

  // 为此缓冲区创建一个RTV
  md3dDevice->CreateRenderTargetView(
    mSwapChainBuffer[i].Get(), nullptr, rtvHeapHandle);

  // 偏移到描述符堆中的下一个缓冲区
  rtvHeapHandle.Offset(1, mRtvDescriptorSize);
```

```
}
```

4.3.8 创建深度/模板缓冲区及其视图

现在来创建程序中所需的深度/模板缓冲区。正如4.1.5节所述，深度缓冲区其实就是一种2D纹理，它存储着离观察者最近的可视对象的深度信息（如果使用了模板，还会附有模板信息）。纹理是一种GPU资源，因此我们要通过填写D3D12_RESOURCE_DESC结构体来描述纹理资源，再用ID3D12Device::CreateCommittedResource方法来创建它。D3D12_RESOURCE_DESC结构体的定义如下[35]。

```
typedef struct D3D12_RESOURCE_DESC
{
  D3D12_RESOURCE_DIMENSION Dimension;
  UINT64 Alignment;
  UINT64 Width;
  UINT Height;
  UINT16 DepthOrArraySize;
  UINT16 MipLevels;
  DXGI_FORMAT Format;
  DXGI_SAMPLE_DESC SampleDesc;
  D3D12_TEXTURE_LAYOUT Layout;
  D3D12_RESOURCE_MTSC_FLAG Misc Flags;
} D3D12_RESOURCE_DESC;
```

1. Dimension：资源的维度，即为下列枚举类型中的成员之一。

```
  enum D3D12_RESOURCE_DIMENSION
  {
    D3D12_RESOURCE_DIMENSION_UNKNOWN = 0,
    D3D12_RESOURCE_DIMENSION_BUFFER = 1,
    D3D12_RESOURCE_DIMENSION_TEXTURE1D = 2,
    D3D12_RESOURCE_DIMENSION_TEXTURE2D = 3,
    D3D12_RESOURCE_DIMENSION_TEXTURE3D = 4
  } D3D12_RESOURCE_DIMENSION;
```

2．Width：以纹素为单位来表示的纹理宽度。对于缓冲区资源来说，此项是缓冲区占用的字节数。

3．Height：以纹素为单位来表示的纹理高度。

4．DepthOrArraySize：以纹素为单位来表示的纹理深度，或者（对于1D纹理和2D纹理来说）是纹理数组的大小。注意，Direct3D中并不存在3D纹理数组的概念。

5．MipLevels：mipmap层级的数量。我们会在第9章讲纹理时介绍mipmap。对于深度/模板缓冲区而言，只能有一个mipmap级别。

6．Format：DXGI_FORMAT枚举类型中的成员之一，用于指定纹素的格式。对于深度/模板缓冲区来说，此格式需要从4.1.5节介绍的格式中选择。

7．SampleDesc：多重采样的质量级别以及对每个像素的采样次数，详情参见4.1.7节和4.1.8节。先来回顾一下4X MSAA技术：为了存储每个子像素的颜色和深度/模板信息，所用后台缓冲区和深度缓冲区的大小要4倍于屏幕的分辨率。因此，深度/模板缓冲区与渲染目标的多重采样设置一定要相匹配。

8．Layout：D3D12_TEXTURE_LAYOUT枚举类型的成员之一，用于指定纹理的布局。我们暂时还不用考虑这个问题，在此将它指定为D3D12_TEXTURE_LAYOUT_UNKNOWN即可。

9. Flags：与资源有关的杂项标志。对于一个深度/模板缓冲区资源来说，要将此项指定为D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL[36]。

GPU资源都存于堆（heap）中，其本质是具有特定属性的GPU显存块。ID3D12Device::CreateCommittedResource方法将根据我们所提供的属性创建一个资源与一个堆，并把该资源提交到这个堆中。

```
HRESULT ID3D12Device::CreateCommittedResource(
  const D3D12_HEAP_PROPERTIES *pHeapProperties,
  D3D12_HEAP_FLAGS HeapFlags,
  const D3D12_RESOURCE_DESC *pDesc,
  D3D12_RESOURCE_STATES InitialResourceState,
  const D3D12_CLEAR_VALUE *pOptimizedClearValue,
  REFIID riidResource,
  void **ppvResource);

typedef struct D3D12_HEAP_PROPERTIES {
 D3D12_HEAP_TYPE       Type;
 D3D12_CPU_PAGE_PROPERTY CPUPageProperty;
 D3D12_MEMORY_POOL     MemoryPoolPreference;
 UINT CreationNodeMask;
 UINT VisibleNodeMask;
} D3D12_HEAP_PROPERTIES[37];
```

1. pHeapProperties：（资源欲提交至的）堆所具有的属性。有一些属性是针对高级用法而设。目前只需关心D3D12_HEAP_PROPERTIES中的D3D12_HEAP_TYPE枚举类型这一主要属性，其中的成员列举如下。

a） D3D12_HEAP_TYPE_DEFAULT：默认堆（default heap）。向这堆里提交的资源，唯独GPU可以访问。举一个有关深度/模板缓冲区的例子：GPU会读写深度/模板缓冲区，而CPU从不需要访问它，所以深

度/模板缓冲区应被放入默认堆中。

b）D3D12_HEAP_TYPE_UPLOAD：上传堆（upload heap）。向此堆里提交的都是需要经CPU上传至GPU的资源。

c）D3D12_HEAP_TYPE_READBACK：回读堆（read-back heap）。向这种堆里提交的都是需要由CPU读取的资源。

d）D3D12_HEAP_TYPE_CUSTOM：此成员应用于高级场景——更多信息可详见MSDN文档。

2．HeapFlags：与（资源欲提交至的）堆有关的额外选项标志。通常将它设为D3D12_HEAP_FLAG_NONE[38]。

3．pDesc：指向一个D3D12_RESOURCE_DESC实例的指针，用它描述待建的资源。

4．InitialResourceState：回顾4.2.3节的内容可知，不管何时，每个资源都会处于一种特定的使用状态。在资源创建时，需要用此参数来设置它的初始状态。对于深度/模板缓冲区来说，通常将其初始状态设置为D3D12_RESOURCE_STATE_COMMON，再利用ResourceBarrier方法辅以D3D12_RESOURCE_STATE_DEPTH_WRITE状态，将其转换为可以绑定在渲染流水线上的深度/模板缓冲区[39]。

5．pOptimizedClearValue：指向一个D3D12_CLEAR_VALUE对象

的指针，它描述了一个用于清除资源的优化值。选择适当的优化清除值，可提高清除操作的执行速度。若不希望指定优化清除值，可把此参数设为nullptr。

```
struct D3D12_CLEAR_VALUE
{
  DXGI_FORMAT Format;
  union
  {
    FLOAT Color[ 4 ];
    D3D12_DEPTH_STENCIL_VALUE DepthStencil;
  };
}     D3D12_CLEAR_VALUE;
```

6. riidResource：我们希望获得的ID3D12Resource接口的COM ID。

7. ppvResource：返回一个指向ID3D12Resource的指针，即新建的资源。

注意 Note

为了使性能达到最佳，通常应将资源放置于默认堆中。只有在需要使用上传堆或回读堆的特性之时，才选用其他类型的堆。

另外，在使用深度/模板缓冲区之前，一定要创建相关的深度/模板视图，并将它绑定到渲染流水线上。这个流程类似于创建渲染目标视

图。下面的代码演示了该如何创建深度/模板纹理及相应的深度/模板视图：

```
// 创建深度/模板缓冲区及其视图
D3D12_RESOURCE_DESC depthStencilDesc;
depthStencilDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
depthStencilDesc.Alignment = 0;
depthStencilDesc.Width = mClientWidth;
depthStencilDesc.Height = mClientHeight;
depthStencilDesc.DepthOrArraySize = 1;
depthStencilDesc.MipLevels = 1;
depthStencilDesc.Format = mDepthStencilFormat;
depthStencilDesc.SampleDesc.Count = m4xMsaaState ? 4 : 1;
depthStencilDesc.SampleDesc.Quality = m4xMsaaState ? (m4xMsaaQuality - 1)
: 0;
depthStencilDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
depthStencilDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL;

D3D12_CLEAR_VALUE optClear;
optClear.Format = mDepthStencilFormat;
optClear.DepthStencil.Depth = 1.0f;
optClear.DepthStencil.Stencil = 0;
ThrowIfFailed(md3dDevice->CreateCommittedResource(
  &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
  D3D12_HEAP_FLAG_NONE,
  &depthStencilDesc,
  D3D12_RESOURCE_STATE_COMMON,
  &optClear,
  IID_PPV_ARGS(mDepthStencilBuffer.GetAddressOf())));

// 利用此资源的格式，为整个资源的第0 mip层创建描述符
md3dDevice->CreateDepthStencilView(
  mDepthStencilBuffer.Get(),
  nullptr,
  DepthStencilView());

// 将资源从初始状态转换为深度缓冲区
mCommandList->ResourceBarrier(
  1,
  &CD3DX12_RESOURCE_BARRIER::Transition(
    mDepthStencilBuffer.Get(),
    D3D12_RESOURCE_STATE_COMMON,
    D3D12_RESOURCE_STATE_DEPTH_WRITE));
```

注意，刚刚采用了CD3DX12_HEAP_PROPERTIES辅助构造函数来创建堆的属性结构体，它的具体实现如下：

```
explicit CD3DX12_HEAP_PROPERTIES(
    D3D12_HEAP_TYPE type,
    UINT creationNodeMask = 1,
    UINT nodeMask = 1 )
{
  Type = type;
  CPUPageProperty = D3D12_CPU_PAGE_PROPERTY_UNKNOWN;
  MemoryPoolPreference = D3D12_MEMORY_POOL_UNKNOWN;
  CreationNodeMask = creationNodeMask;
  VisibleNodeMask = nodeMask;
}
```

CreateDepthStencilView方法的第二个参数是指向D3D12_DEPTH_STENCIL_VIEW_DESC结构体的指针。这个结构体描述了资源中元素的数据类型（格式）。如果资源在创建时已指定了具体格式（即此资源不是无类型格式），那么就可以把该参数设为空指针，表示以该资源创建时的格式为它的第一个mipmap层级创建一个视图（在创建深度/模板缓冲区时就只有一个mipmap层级，mipmap的相关知识将在第9章中进行讨论）。由于我们已经为深度/模板缓冲区设置了具体格式，所以向此参数传入空指针。

4.3.9 设置视口

我们通常会将3D场景绘制到与整个屏幕（在全屏模式下）或整个窗口工作区大小相当的后台缓冲区中。但是，有时只是希望把3D场景绘制到后台缓冲区的某个矩形子区域当中，如图4.9所示。

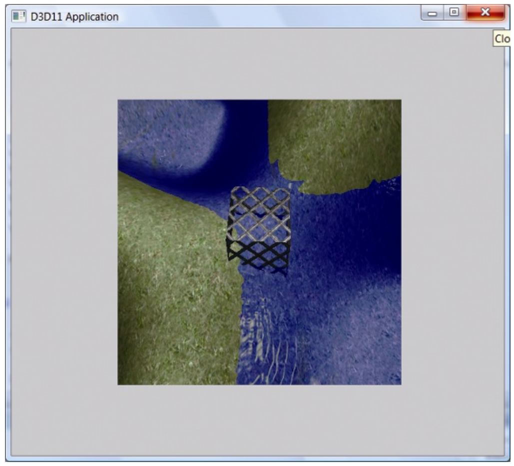


图4.9　通过修改视口，我们就能将3D场景绘制到后台缓冲区内的矩形子区域当中，继而使后台缓冲区中的内容呈现在窗口的工作区范围之内

我们把后台缓冲区中的这种矩形子区域叫作视口（viewport），并通过下列结构体来描述它：

```
typedef struct D3D12_VIEWPORT {
  FLOAT TopLeftX;
  FLOAT TopLeftY;
  FLOAT Width;
  FLOAT Height;
  FLOAT MinDepth;
  FLOAT MaxDepth;
} D3D12_VIEWPORT;
```

结构体中的前4个数据成员定义了视口矩形（viewport rectangle）相对于后台缓冲区的绘制范围（由于数据成员是用`float`类型表示的，所以我们能够以小数精度来指定像素坐标）[40]。在Direct3D中，存储在深度缓冲区中的数据都是范围在0～1的归一化深度值。`MinDepth`和`MaxDepth`这两个成员负责将深度值从区间[0, 1]转换到区间[`MinDepth`, `MaxDepth`]。通过对深度范围进行转换即可实现某些特效，例如，我们可以依次设置`MinDepth=0`和`MaxDepth=0`，用此视口绘制的物体其深度

值都为0，它们将比场景中其他物体的位置都更靠前。然而，在大多数情况下通常会把MinDepth与MaxDepth分别设置为0与1，也就是令深度值保持不变。

只要填写好D3D12_VIEWPORT结构体， 便可以用ID3D12GraphicsCommandList::RSSetViewports方法来设置Direct3D中的视口了。下面的示例是通过创建并设置一个视口，将场景绘至整个后台缓冲区：

```
D3D12_VIEWPORT vp;
vp.TopLeftX = 0.0f;
vp.TopLeftY = 0.0f;
vp.Width  = static_cast<float>(mClientWidth);
vp.Height  = static_cast<float>(mClientHeight);
vp.MinDepth = 0.0f;
vp.MaxDepth = 1.0f;

mCommandList->RSSetViewports(1, &vp);
```

第一个参数是要绑定的视口数量（有些高级效果需要使用多个视口），第二个参数是一个指向视口数组的指针。

注意 Note

不能为同一个渲染目标指定多个视口。而多视口（multiple viewport）则是一种用于对多个渲染目标同时进行渲染的高级技术。

注意 Note

命令列表一旦被重置，视口也就需要随之而重置。

事实上，还可以用视口技术来实现双人游戏的分屏（split screen）模式。首先创建两个视口，一个占屏幕左半部，另一个占右半部。接下来，在左视口中以玩家1的视角来绘制3D场景，再在右视口中以玩家2的视角来绘制3D场景即可。

4.3.10　设置裁剪矩形

我们可以相对于后台缓冲区定义一个裁剪矩形（scissor rectangle），在此矩形外的像素都将被剔除（即这些图像部分将不会被光栅化（rasterize）至后台缓冲区）。这个方法能用于优化程序的性能。例如，假设已知有一个矩形的UI（user interface，用户界面）元素覆于屏幕中某块区域的最上层，那么我们也就无须对3D空间中那些被它遮挡的像素进行处理了。

裁剪矩形由类型为RECT的D3D12_RECT结构体（typedef RECT D3D12_RECT;）定义而成：

```
typedef struct tagRECT
{
  LONG  left;
  LONG  top;
  LONG  right;
  LONG  bottom;
} RECT;
```

在Direct3D中，要用ID3D12GraphicsCommandList::RSSetScissorRects方法来设置裁剪矩形。下面的示例将创建并设置一个覆盖后台缓冲区左上角1/4区域的裁剪矩形：

```
mScissorRect = { 0, 0, mClientWidth/2, mClientHeight/2 };
mCommandList->RSSetScissorRects(1, &mScissorRect);
```

类似于RSSetViewports方法，RSSetScissorRects方法的第一个参数是要绑定的裁剪矩形数量（为了实现一些高级效果有时会采用多个裁剪矩形），第二个参数是指向一个裁剪矩形数组的指针。

注意 Note

不能为同一个渲染目标指定多个裁剪矩形。多裁剪矩形（multiple scissor rectangle）是一种用于同时对多个渲染目标进行渲染的高级技术。

注意 Note

裁剪矩形需要随着命令列表的重置而重置。

4.4　计时与动画

为了制作出精准的动画效果就需要精确地计量时间，特别是要准确地度量出动画每帧画面之间的时间间隔。如果帧率（frame rate，也有作帧速率、帧频等，每秒刷新的帧数）较高，那么帧与帧之间的间隔就会比较短，此时我们就要用到高精度的计时器。

4.4.1　性能计时器

为了精确地度量时间，我们将采用性能计时器（performance timer。或称性能计数器，performance counter）。如果希望调用查询性能计时器的Win32函数，我们必须引入头文件`#include <windows.h>`。

性能计时器所用的时间度量单位叫作计数（count）。可调用`QueryPerformanceCounter`函数来获取性能计时器测量的当前时刻值（以计数为单位）：

```
__int64 currTime;
QueryPerformanceCounter((LARGE_INTEGER*)&currTime);
```

观察可知，此函数通过参数返回的当前时刻值是个64位的整数。

再用`QueryPerformanceFrequency`函数来获取性能计时器的频率（单位：计数/秒）：

```
__int64 countsPerSec;
QueryPerformanceFrequency((LARGE_INTEGER*)&countsPerSec);
```

每个计数所代表的秒数（或称几分之一秒），即为上述性能计时器频率的倒数：

```
mSecondsPerCount = 1.0 / (double)countsPerSec;
```

因此，只需将读取的时刻计数值valueInCounts乘以转换因子mSecondsPerCount，就可以将其单位转换为秒：

```
valueInSecs = valueInCounts * mSecondsPerCount;
```

对我们而言，单次调用QueryPerformanceCounter函数所返回的时刻值并没有什么特别的意义。如果隔一小段时间，再调用一次该函数，并得到此时的时刻值，我们就会发现这两次调用的时刻间隔即为两个返回值的差。因此，我们总是以两个时间戳（time stamp）的相对差值，而非性能计数器单次返回的实际值来度量时间。通过下面的代码来明确这一想法：

```
__int64 A = 0;
QueryPerformanceCounter((LARGE_INTEGER*)&A);

// 执行预定的逻辑

__int64 B = 0;
QueryPerformanceCounter((LARGE_INTEGER*)&B);
```

利用(B-A)即可获得代码执行期间的计数值，或以(B-A)*mSecondsPerCount获取代码运行期间所花费的秒数[41]。

注意 Note

MSDN对QueryPerformanceCounter函数作有如下备注："按道理来讲，对于一台具有多个处理器的计算机而言，无论在哪一个处理器上调用此函数都应返回当前时刻的计数值。然而，由于基本输入/输出系统（BIOS）或硬件抽象层（HAL）上的缺陷，导致了在不同的处理器上可能会得到不同的结果[42]。"对此，我们可以通过SetThreadAffinityMask函数，防止应用程序的主线程切换到其他的处理器上去执行指令，从而实现每次都能在同一处理器上两次调用QueryPerformanceCounter函数，得到正确的计数差值。

4.4.2 游戏计时器类

在接下来的两小节中，我们将讨论以下GameTimer类的实现：

```
class GameTimer
{
public:
    GameTimer();

    float TotalTime()const; // 用秒作为单位
    float DeltaTime()const; // 用秒作为单位

    void Reset(); // 在开始消息循环之前调用
    void Start(); // 解除计时器暂停时调用
    void Stop();  // 暂停计时器时调用
    void Tick();  // 每帧都要调用
```

```
private:
    double mSecondsPerCount;
    double mDeltaTime;

    __int64 mBaseTime;
    __int64 mPausedTime;
    __int64 mStopTime;
    __int64 mPrevTime;
    __int64 mCurrTime;

    bool mStopped;
};
```

此类的构造函数会查询性能计数器的频率。另外几个成员函数将在后面的两小节中讨论。

```
GameTimer::GameTimer()
: mSecondsPerCount(0.0), mDeltaTime(-1.0), mBaseTime(0),
 mPausedTime(0), mPrevTime(0), mCurrTime(0), mStopped(false)
{
    __int64 countsPerSec;
    QueryPerformanceFrequency((LARGE_INTEGER*)&countsPerSec);
    mSecondsPerCount = 1.0 / (double)countsPerSec;
}
```

`GameTimer`类的实现位于`GameTimer.h`和`GameTimer.cpp`文件之中，可以在本书源代码的Common目录里找到。

4.4.3　帧与帧之间的时间间隔

当渲染动画帧时，我们需要知道每帧之间的时间间隔，以此来根据时间的流逝对游戏对象进行更新。计算帧与帧之间间隔的流程如下。假设在开始显示第i帧画面时，性能计数器返回的时刻为t_i；而此前的一帧开始显示时，性能计数器返回的时刻为t_{i-1}。那么，这两帧的时间间隔

就是$\Delta t = t_i - t_{i-1}$。对于实时渲染来说，为了保证动画的流畅性至少需要每秒刷新30帧（实际上通常会采用更高的帧率），所以$\Delta t = t_i - t_{i-1}$往往是个较小的数值。

计算Δt的代码如下：

```
void GameTimer::Tick()
{
    if( mStopped )
    {
        mDeltaTime = 0.0;
        return;
    }

    // 获得本帧开始显示的时刻
    __int64 currTime;
    QueryPerformanceCounter((LARGE_INTEGER*)&currTime);
    mCurrTime = currTime;

    // 本帧与前一帧的时间差
    mDeltaTime = (mCurrTime - mPrevTime)*mSecondsPerCount;

    // 准备计算本帧与下一帧的时间差
    mPrevTime = mCurrTime;

    // 使时间差为非负值。DXSDK中的CDXUTTimer示例注释里提到：如果处理器处于节能
模式，或者在
    // 计算两帧间时间差的过程中切换到另一个处理器时（即QueryPerformanceCounter
函数的两次调
    // 用并非在同一处理器上），则mDeltaTime有可能会成为负值
    if(mDeltaTime < 0.0)
    {
        mDeltaTime = 0.0;
    }
}

float GameTimer::DeltaTime()const
{
    return (float)mDeltaTime;
}
```

Tick函数被调用于程序的消息循环之中：

```
int D3DApp::Run()
{
  MSG msg = {0};

  mTimer.Reset();

  while(msg.message != WM_QUIT)
  {
    // 如果有窗口消息就进行处理
    if(PeekMessage( &msg, 0, 0, 0, PM_REMOVE ))
    {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }
    // 否则就执行动画与游戏的相关逻辑
    else
    {
      mTimer.Tick();

      if( !mAppPaused )
      {
        CalculateFrameStats();
        Update(mTimer);
        Draw(mTimer);
      }
      else
      {
        Sleep(100);
      }
    }
  }

  return (int)msg.wParam;
}
```

采用这种方案时，我们需要在每一帧都计算Δt，并将其送入Update方法。只有这样，才可以根据前一动画帧所花费的时间对场景进行更新。以下是Reset方法的具体实现：

```
void GameTimer::Reset()
{
    __int64 currTime;
```

```
    QueryPerformanceCounter((LARGE_INTEGER*)&currTime);

    mBaseTime = currTime;
    mPrevTime = currTime;
    mStopTime = 0;
    mStopped = false;
}
```

这段代码内的一些变量还未曾讨论（参见4.4.4节）。但是可以看出，在调用`Reset`时会将`mPrevTime`初始化为当前时刻。这一步十分关键，由于在第一帧画面之前没有任何的动画帧，所以此帧的前一个时间戳t_{i-1}并不存在。因此，在消息循环开始之前，需要通过`Reset`方法对`mPrevTime`的值进行初始化。

4.4.4 总时间

除此之外，我们还开展了一项名为总时间（total time）的实用时间统计：这是一种自应用程序开始，不计其中暂停时间的时间总和。下面的情景展示了它所起到的作用。假设我们制作的游戏需要玩家在300秒内打通一个关卡。关卡开始时，先来获取时间t_{start}，它表示自程序开始至此关卡开始所经过的时间。在关卡开始后，我们会经常检测由程序开始至当前的时间t。如果$t - t_{start} > 300$秒（见图4.10），就说明玩家在此关卡停留的时间已经超过300秒，挑战失败。不难发现，在这种情景中，我们并不希望把玩家在游戏过程中的暂停时间也统计在关卡停留的时间内，而总时间则刚好可以满足这一点。

总时间的另一个应用情景是：驱使某量随时间函数而变化。举个例子，假设需要根据某个时间函数来得到光源环绕场景的运动轨迹，那么

它的位置可以用下列参数方程表示：

$$\begin{cases} x = 10\cos t \\ y = 20 \\ z = 10\sin t \end{cases}$$

这里的t表示时间，随着t（时间）的增加，光源的坐标也在不断更新，从而使它在$y = 20$这一平面内半径为10的圆周上运动。对于这种动画，我们也不希望把暂停时间记录在变化时间内，如图4.11所示。

为了统计总时间，我们将使用下列变量：

```
__int64 mBaseTime;
__int64 mPausedTime;
__int64 mStopTime;
```

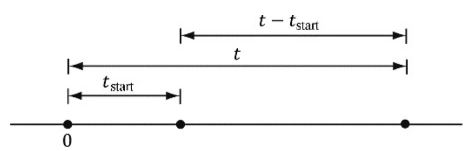


图4.10　计算自游戏关卡开始至当前的时间。注意，我们将此应用程序的开始时刻选择为坐标原点(0)，并以此作为参考系来统计时间

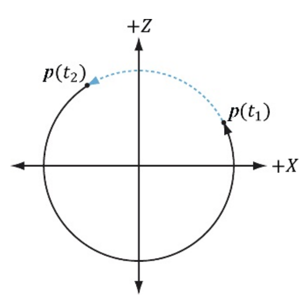


图4.11　如果我们在t_1时暂停计时器，在t_2时重启计时器，并把暂停时间记录在内，那么用户就会看到光点从$\boldsymbol{p}(t_1)$突然跳到$\boldsymbol{p}(t_2)$处

正如4.4.3节中所述，在调用Reset函数之时，会将mBaseTime初始

化为当前时刻。我们可以把这个时刻当作应用程序的开始时刻。在大多数情况下，Reset函数只会在消息循环开始之前调用一次，所以在应用程序的整个生命周期里，mBaseTime一般会保持不变。变量mPausedTime存储的是所有暂停时间之和。这个累积时间很有存在的必要：为了得到不统计暂停时间的总时间，我们可以用程序的总运行时间减去这个累加时间算出。变量mStopTime会给出计时器停止（暂停）[43]的时刻，借此即可记录暂停的时间。

Stop和Start是GameTimer类中的两个关键方法。当应用程序分别处于暂停或未暂停的状态时，我们就可以依情况调用它们，以此令GameTimer能够记录暂停的时间。代码注释介绍了这两个方法的相关细节：

```
void GameTimer::Stop()
{
    // 如果已经处于停止状态，那就什么也不做
    if( !mStopped )
    {
        __int64 currTime;
        QueryPerformanceCounter((LARGE_INTEGER*)&currTime);

        // 否则，保存停止的时刻，并设置布尔标志，指示计时器已经停止
        mStopTime = currTime;
        mStopped = true;
    }
}

void GameTimer::Start()
{
    __int64 startTime;
    QueryPerformanceCounter((LARGE_INTEGER*)&startTime);

    // 累加调用stop和start这对方法之间的暂停时刻间隔
    //
    //                         |<-------d------->|
    // ----*---------------*-----------------*------------> 时间
```

```
    // mBase Time        mStopTime            startTime

    // 如果从停止状态继续计时的话……
    if( mStopped )
    {
        // 累加暂停时间
        mPausedTime += (startTime - mStopTime);

        // 在重新开启计时器时，前一帧的时间mPrevTime是无效的，这是因为它存储的是暂停时前一
        // 帧的开始时刻，因此需要将它重置为当前时刻

        mPrevTime = startTime;

        // 已不再是停止状态……
        mStopTime = 0;
        mStopped = false;
    }
}
```

最后，我们就可以用成员函数TotalTime返回自调用Reset函数开始不计暂停时间的总时间了，它的具体实现如下：

```
float GameTimer::TotalTime()const
{
// 如果正处于停止状态，则忽略本次停止时刻至当前时刻的这段时间。此外，如果之前已有过暂停的情况，
// 那么也不应统计mStopTime – mBaseTime这段时间内的暂停时间
// 为了做到这一点，可以从mStopTime中再减去暂停时间mPausedTime
//
//                              前一次暂停时间                  当前的暂停时间
//                           |<--------------->|           |<---------->|
// ----*---------------*-----------------*------------*------------*------> 时间
// mBase Time      mStopTime0           startTime    mStopTime     mCurrTime

    if( mStopped )
    {
        return (float)(((mStopTime - mPausedTime)-
                mBaseTime)*mSecondsPerCount);
    }

// 我们并不希望统计mCurrTime – mBaseTime内的暂停时间
// 可以通过从mCurrTime中再减去暂停时间mPausedTime来实现这一点
```

```
//
// (mCurrTime - mPausedTime) - mBaseTime
//
//                        |<--  暂停时间  -->|
// ----*---------------*-----------------*------------*------> 时间
// mBaseTime        mStopTime          startTime     mCurrTime

    else
    {
        return (float)(((mCurrTime-mPausedTime)-
                mBaseTime)*mSecondsPerCount);
    }
}
```

在我们的演示框架中，为了度量从程序开始到某时刻的总时间而创建了一个GameTimer实例，与此同时，也对每帧之间的时间间隔进行了测量。其实，我们也可以再创建一个GameTimer实例，把它当作一个通用的“秒表”。例如，当游戏中的炸弹被点燃时，我们可以开启一个新的GameTimer实例，当TotalTime达到5秒时就触发爆炸事件。

4.5　应用程序框架示例

全书的演示程序都使用了d3dUtil.h、d3dUtil.cpp、d3dApp.h和d3dApp.cpp中的框架代码，可以从本书的官方网站下载到这些文件。d3dUtil.h和d3dUtil.cpp文件中含有程序所需的实用工具代码，d3dApp.h和d3dApp.cpp文件内包含用于封装Direct3D示例程序的Direct3D应用程序类核心代码。由于在书中不能细述这些文件里的每一行代码（例如，我们不会展示如何创建一个窗口，基本的Win32编程是阅读本书的必备知识[44]），所以我们鼓励读者在阅读完本章后，仔细研究这些文件。构建此框架的目标是：隐去窗口创建和Direct3D初始化的具体细节；通过对这些代码进行封装，那些细枝末节就不会分散我们的注意力，继而使我们把精力集中在重点代码上。

4.5.1　D3DApp类

D3DApp类是一种基础的Direct3D应用程序类，它提供了创建应用程序主窗口、运行程序消息循环、处理窗口消息以及初始化Direct3D等多种功能的函数。此外，该类还为应用程序例程定义了一组框架函数。我们可以根据需求通过实例化一个继承自D3DApp的类，重写（override）框架的虚函数，以此从D3DApp类中派生出自定义的用户代码。D3DApp类的定义如下：

```
#include "d3dUtil.h"
#include "GameTimer.h"
```

```
// 链接所需的d3d12库
#pragma comment(lib,"d3dcompiler.lib")
#pragma comment(lib, "D3D12.lib")
#pragma comment(lib, "dxgi.lib")

class D3DApp
{
protected:

  D3DApp(HINSTANCE hInstance);
  D3DApp(const D3DApp& rhs) = delete;
  D3DApp& operator=(const D3DApp& rhs) = delete;
  virtual ~D3DApp();

public:

  static D3DApp* GetApp();

  HINSTANCE AppInst()const;
  HWND   MainWnd()const;
  float   AspectRatio()const;

  bool Get4xMsaaState()const;
  void Set4xMsaaState(bool value);

  int Run();

  virtual bool Initialize();
  virtual LRESULT MsgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lPara
m);

protected:
  virtual void CreateRtvAndDsvDescriptorHeaps();
  virtual void OnResize();
  virtual void Update(const GameTimer& gt)=0;
  virtual void Draw(const GameTimer& gt)=0;

  // 便于重写鼠标输入消息的处理流程
  virtual void OnMouseDown(WPARAM btnState, int x, int y){ }
  virtual void OnMouseUp(WPARAM btnState, int x, int y) { }
  virtual void OnMouseMove(WPARAM btnState, int x, int y){ }

protected:

  bool InitMainWindow();
  bool InitDirect3D();
  void CreateCommandObjects();
  void CreateSwapChain();
```

```
  void FlushCommandQueue();

  ID3D12Resource* CurrentBackBuffer()const
  {
    return mSwapChainBuffer[mCurrBackBuffer].Get();
  }

  D3D12_CPU_DESCRIPTOR_HANDLE CurrentBackBufferView()const
  {
    return CD3DX12_CPU_DESCRIPTOR_HANDLE(
      mRtvHeap->GetCPUDescriptorHandleForHeapStart(),
      mCurrBackBuffer,
      mRtvDescriptorSize);
  }

  D3D12_CPU_DESCRIPTOR_HANDLE DepthStencilView()const
  {
    return mDsvHeap->GetCPUDescriptorHandleForHeapStart();
  }

  void CalculateFrameStats();

  void LogAdapters();
  void LogAdapterOutputs(IDXGIAdapter* adapter);
  void LogOutputDisplayModes(IDXGIOutput* output, DXGI_FORMAT format);

protected:

  static D3DApp* mApp;

  HINSTANCE mhAppInst = nullptr; // 应用程序实例句柄
  HWND   mhMainWnd = nullptr; // 主窗口句柄
  bool   mAppPaused = false; // 应用程序是否暂停
  bool   mMinimized = false; // 应用程序是否最小化
  bool   mMaximized = false; // 应用程序是否最大化
  bool   mResizing = false;  // 大小调整栏是否受到拖拽
  bool   mFullscreenState = false;// 是否开启全屏模式

  // 若将该选项设置为true，则使用4X MSAA技术(参见4.1.8节)。默认值为false
  bool   m4xMsaaState = false;  // 是否开启4X MSAA
  UINT   m4xMsaaQuality = 0;    // 4X MSAA的质量级别

  // 用于记录"delta-time"（帧之间的时间间隔）和游戏总时间(参见4.4节)
  GameTimer mTimer;

  Microsoft::WRL::ComPtr<IDXGIFactory4> mdxgiFactory;
```

```
  Microsoft::WRL::ComPtr<IDXGISwapChain> mSwapChain;
  Microsoft::WRL::ComPtr<ID3D12Device> md3dDevice;

  Microsoft::WRL::ComPtr<ID3D12Fence> mFence;
  UINT64 mCurrentFence = 0;

  Microsoft::WRL::ComPtr<ID3D12CommandQueue> mCommandQueue;
  Microsoft::WRL::ComPtr<ID3D12CommandAllocator> mDirectCmdListAlloc;
  Microsoft::WRL::ComPtr<ID3D12GraphicsCommandList> mCommandList;

  static const int SwapChainBufferCount = 2;
  int mCurrBackBuffer = 0;
  Microsoft::WRL::ComPtr<ID3D12Resource> mSwapChainBuffer[SwapChainBufferC
ount];
  Microsoft::WRL::ComPtr<ID3D12Resource> mDepthStencilBuffer;

  Microsoft::WRL::ComPtr<ID3D12DescriptorHeap> mRtvHeap;
  Microsoft::WRL::ComPtr<ID3D12DescriptorHeap> mDsvHeap;

  D3D12_VIEWPORT mScreenViewport;
  D3D12_RECT mScissorRect;

  UINT mRtvDescriptorSize = 0;
  UINT mDsvDescriptorSize = 0;
  UINT mCbvSrvUavDescriptorSize = 0;

  // 用户应该在派生类的派生构造函数中自定义这些初始值
  std::wstring mMainWndCaption = L"d3d App";
  D3D_DRIVER_TYPE md3dDriverType = D3D_DRIVER_TYPE_HARDWARE;
  DXGI_FORMAT mBackBufferFormat = DXGI_FORMAT_R8G8B8A8_UNORM;
  DXGI_FORMAT mDepthStencilFormat = DXGI_FORMAT_D24_UNORM_S8_UINT;
  int mClientWidth = 800;
  int mClientHeight = 600;
};
```

我们在上面的代码中用注释描述了一些数据成员，其中的方法将陆续在后面的章节中讨论。

4.5.2 非框架方法

下面列举几种常用的非框架方法。

1．`D3DApp`：这个构造函数只是简单地将数据成员初始化为默认值。

2．`~D3DApp`：这个析构函数用于释放`D3DApp`中所用的COM接口对象并刷新命令队列。在析构函数中刷新命令队列的原因是：在销毁GPU引用的资源以前，必须等待GPU处理完队列中的所有命令。否则，可能造成应用程序在退出时崩溃。

```
D3DApp::~D3DApp()
{
  if(md3dDevice != nullptr)
    FlushCommandQueue();
}
```

3．`AppInst`：简单的存取函数，返回应用程序实例句柄。

4．`MainWnd`：简单的存取函数，返回主窗口句柄。

5．`AspectRatio`：这个纵横比（亦有译作长宽比、宽高比[45]）定义的是后台缓冲区的宽度与高度之比。第5章会用到这个比值。它的实现比较简单：

```
  float D3DApp::AspectRatio()const
  {
    return static_cast<float>(mClientWidth) / mClientHeight;
  }
```

6．`Get4xMsaaState`：如果启用4X MSAA就返回true，否则返回false。

7．`Set4xMsaaState`：开启或禁用4X MSAA功能。

8. Run：这个方法封装了应用程序的消息循环。它使用的是Win32的PeekMessage函数，当没有窗口消息到来时就会处理我们的游戏逻辑部分。该方法的实现可见4.4.3节。

9. InitMainWindow：初始化应用程序主窗口。本书假设读者熟悉基本的Win32窗口初始化流程。

10. InitDirect3D：通过实现4.3节中讨论的步骤来完成Direct3D的初始化。

11. CreateSwapChain：创建交换链（参见4.3.5节）。

12. CreateCommandObjects：依4.3.4节中所述的流程创建命令队列、命令列表分配器和命令列表。

13. FlushCommandQueue：强制CPU等待GPU，直到GPU处理完队列中所有的命令（详见4.2.2节）。

14. CurrentBackBuffer：返回交换链中当前后台缓冲区的ID3D12Resource。

15. CurrentBackBufferView：返回当前后台缓冲区的RTV（渲染目标视图，render target view）。

16. DepthStencilView：返回主深度/模板缓冲区的DSV（深度/模板视图，depth/stencil view）。

17．CalculateFrameStats：计算每秒的平均帧数以及每帧平均的毫秒时长。实现方法将在4.5.4节中讨论。

18．LogAdapters：枚举系统中所有的适配器（参见4.1.10节）。

19．LogAdapterOutputs：枚举指定适配器的全部显示输出（参见4.1.10节）。

20．LogOutputDisplayModes：枚举某个显示输出对特定格式支持的所有显示模式（参见4.1.10节）。

4.5.3　框架方法

对于本书的所有示例程序来说，我们每次都会重写D3DApp中的6个虚函数。这6个函数用于针对特定的示例来实现所需的具体功能。这种设定的好处是把初始化代码、消息处理等流程都统一实现在D3DApp类中，继而使我们可以把精力集中在特定例程中的关键代码之上。以下是对这6个框架方法的概述。

1．Initialize：通过此方法为程序编写初始化代码，例如分配资源、初始化对象和建立3D场景等。D3DApp类实现的初始化方法会调用InitMainWindow和InitDirect3D，因此，我们在自己实现的初始化派生方法中，应当首先像下面那样来调用D3DApp类中的初始化方法：

```
bool TestApp::Initialize()
```

```
{
  if(!D3DApp::Initialize ())
    return false;

  /* 其他的初始化代码请置于此 */
}
```

这样一来，我们的初始化代码才能访问到D3DApp类中的初始化成员。

2. MsgProc：该方法用于实现应用程序主窗口的窗口过程函数（procedure function）。一般来说，如果需要处理在D3DApp::MsgProc中没有得到处理（或者不能如我们所愿进行处理）的消息，只要重写此方法即可。该方法的实现在4.5.5节中有相应的讲解。此外，如果对该方法进行了重写，那么其中并未处理的消息都应当转交至D3DApp::MsgProc。

3. CreateRtvAndDsvDescriptorHeaps：此虚函数用于创建应用程序所需的RTV和DSV描述符堆。默认的实现是创建一个含有SwapChainBufferCount个RTV描述符的RTV堆（为交换链中的缓冲区而创建），以及具有一个DSV描述符的DSV堆（为深度/模板缓冲区而创建）。该方法的默认实现足以满足大多数的示例，但是，为了使用多渲染目标（multiple render targets）这种高级技术，届时仍将重写此方法。

4. OnResize：当D3DApp::MsgProc函数接收到WM_SIZE消息时便会调用此方法。若窗口的大小发生了改变，一些与工作区大小有关的Direct3D属性也需要随之调整。特别是后台缓冲区以及深度/模板缓冲

区，为了匹配窗口工作区调整后的大小需要对其重新创建。我们可以通过调用`IDXGISwapChain::ResizeBuffers`方法来调整后台缓冲区的尺寸。对于深度/模板缓冲区而言，则需要在销毁后根据新的工作区大小进行重建。另外，渲染目标和深度/模板的视图也应重新创建。`D3DApp`类中`OnResize`方法实现的功能即为调整后台缓冲区和深度/模板缓冲区的尺寸，我们可直接查阅其源代码来研究相关细节。除了这些缓冲区以外，依赖于工作区大小的其他属性（如投影矩阵，projection matrix）也要在此做相应的修改。由于在调整窗口大小时，客户端代码可能还需执行一些它自己的逻辑代码，因此该方法亦属于框架的一部分。

5. `Update`：在绘制每一帧时都会调用该抽象方法，我们通过它来随着时间的推移而更新3D应用程序（如呈现动画、移动摄像机、做碰撞检测以及检查用户的输入等）。

6. `Draw`：在绘制每一帧时都会调用的抽象方法。我们在该方法中发出渲染命令，将当前帧真正地绘制到后台缓冲区中。当完成帧的绘制后，再调用`IDXGISwapChain::Present`方法将后台缓冲区的内容显示在屏幕上。

注意　Note

除了上述6个框架方法之外，我们为了便于处理鼠标的按下、释放和移动事件，还分别提供了3个相关的虚函数：

```
virtual void OnMouseDown(WPARAM btnState, int x, int y){ }
virtual void OnMouseUp(WPARAM btnState, int x, int y) { }
virtual void OnMouseMove(WPARAM btnState, int x, int y){ }
```

如此一来，若希望处理鼠标消息，我们只需重写这几种方法，而不必重写MsgProc方法。这3个处理鼠标消息方法的第一个参数都是WPARAM，它存储了鼠标按键的状态（即鼠标事件发生时，哪个键被按下）。第二个和第三个参数则表示鼠标指针在工作区的坐标(x, y)。

4.5.4　帧的统计信息

游戏和图形应用程序往往都会测量每秒渲染的帧数（frames per second，FPS）作为一种画面流畅度的标杆。为此，我们仅需统计在特定时段t内所处理的帧数（并将帧数存于变量n中）即可。因此，时段t内的平均FPS值即为$fps_{avg} = n/t$。如果设$t = 1$，那么$fps_{avg} = n/1 = n$。在我们的代码中，实际所用的时段就是$t = 1$（秒），这样做可省去一次除法运算。再者，用1秒为限会取到一个比较合理的平均值——这段时间不长不短，刚好合适。D3DApp::CalculateFrameStats方法提供了计算FPS的相关代码：

```
void D3DApp::CalculateFrameStats()
{
    // 这段代码计算了每秒的平均帧数，也计算了每帧的平均渲染时间
    // 这些统计值都会被附加到窗口的标题栏中

    static int frameCnt = 0;
    static float timeElapsed = 0.0f;
```

```
    frameCnt++;

    // 以1秒为统计周期来计算平均帧数以及每帧的平均渲染时间
    if( (mTimer.TotalTime() - timeElapsed) >= 1.0f )
    {
        float fps = (float)frameCnt; // fps = frameCnt / 1
        float mspf = 1000.0f / fps;

        wstring fpsStr = to_wstring(fps);
        wstring mspfStr = to_wstring(mspf);

        wstring windowText = mMainWndCaption +
           L"  fps: " + fpsStr +
           L"  mspf: " + mspfStr;
           SetWindowText(mhMainWnd, windowText.c_str());

           // 为计算下一组平均值而重置
           frameCnt = 0;
           timeElapsed += 1.0f;
    }
}
```

为了统计帧数，在每一帧中都要调用此方法。

除了计算FPS外，以上代码也统计了渲染一帧所花费的平均时间（以毫秒计）：

```
float mspf = 1000.0f / fps;
```

每帧所花费的秒数即FPS的倒数，我们可通过将此倒数乘以1000 ms / 1 s来把单位从秒转换到毫秒（1s为1000ms）。

这一行代码的意思是计算渲染一帧画面所花费的毫秒数，这是一种与FPS截然不同的统计量（但是此值可由FPS推导出来）。事实上，知道渲染一帧所花费的时间要比了解FPS更为有效，因为随着场景的转换，我们通过前者就能直观地看出每一帧渲染时长的增减。而FPS却不

能在场景改变后立即反映出渲染时间的变化。此外，正如[Dunlop03]在《FPS versus Frame Time（FPS vs. 帧时间）》一文中所指出的，由于FPS曲线图（FPS curve）的非线性特征，使得采用FPS进行分析可能会得到误导性的结果。例如，请考虑场景（一）：假设我们的应用程序跑到了1000 FPS，利用1 ms（毫秒）就可以渲染1帧。那么，当帧率降到了250 FPS时，渲染1帧就要用4 ms。现在再来思考情景（二）：设想我们的应用程序跑到100 FPS，花10 ms渲染1帧。如果帧率降到了76.9 FPS，那么渲染1帧将花费约13 ms。这两种情景中每帧的渲染时间都增加了3 ms，也就表示它们在渲染每一帧的过程中都增加了同样多的时间。然而，FPS所反映出的统计值却并不直观。虽然从1000 FPS跌到250 FPS看起来要比从100 FPS下降到76.9 FPS的幅度大得多，但诚如我们所看到的，它们渲染每帧所增加的时间实际上却是相同的。

4.5.5 消息处理函数

我们对应用程序框架中的窗口过程进行了大量的简化工作。在一般情况下，本书的程序并不会过多地涉及Win32消息。事实上，应用程序的核心代码都是在空闲处理期间（即没有窗口消息可处理时）执行的。但是，仍有一些重要的消息需要我们亲自去处理。鉴于窗口过程代码的篇幅，我们并不打算将所有的代码都罗列于此，而是仅解释处理这些消息背后的动机。由于应用框架是本书所有示例的基石，所以我们鼓励读者下载其源代码文件，并花费些时间来熟悉它。

我们要处理的第一个消息是`WM_ACTIVATE`。当一个程序被激活

（activate）或进入非活动状态（deactivate）时便会发送此消息。我们以下列方式来对它进行处理：

```
case WM_ACTIVATE:
  if( LOWORD(wParam) == WA_INACTIVE )
  {
    mAppPaused = true;
    mTimer.Stop();
  }
  else
  {
    mAppPaused = false;
    mTimer.Start();
  }
  return 0;
```

如您所见，当程序变为非活动状态时，我们会将数据成员mAppPaused设置为true，而当程序被激活时，则把数据成员mAppPaused设置为false。另外，当暂停使用应用程序时，我们就停止计时器，一旦程序被再次激活，再令计时器继续工作。如果回顾D3DApp::Run（4.4.3节）的实现，我们会发现：当程序暂停时，将不会再执行后续更新场景的代码，而是把空闲出来的CPU周期返还给操作系统。这样一来，我们的程序就不会在非活动的状态中占用CPU资源了。

下一个要处理的消息是WM_SIZE。前文曾提到过，当用户调整窗口的大小时便会产生此消息。处理这个消息的主要目的是：我们希望使后台缓冲区和深度/模板缓冲区的大小与工作区矩形范围的大小保持一致（从而使图像不会发生拉伸的现象）。所以，在每一次调整窗口大小的时候，我们都要记住改变缓冲区的尺寸。调整缓冲区尺寸的代码实现于

D3DApp::OnResize方法之中。正如前文中所述，调用IDXGISwapChain::ResizeBuffers方法即可改变后台缓冲区的尺寸。而深度/模板缓冲区则需要在销毁之后，根据新的窗口尺寸来重新创建。此外，渲染目标和深度/模板的视图也需随之重建。对于用户拖动调整栏的操作，我们一定要小心对待，因为这个行为会连续发出WM_SIZE消息，但我们不希望随之连续调整缓冲区。因此，如若确定用户正在拖动边框调整窗口大小，我们理应什么也不做（暂停应用程序除外），直到用户完成调整操作后再执行修改缓冲区等操作。通过处理WM_EXITSIZEMOVE消息就可以实现这一点。这条消息会在用户释放调整栏时发送。

```
// 当用户抓取调整栏时发送WM_ENTERSIZEMOVE消息
case WM_ENTERSIZEMOVE:
  mAppPaused = true;
  mResizing = true;
  mTimer.Stop();
  return 0;

// 当用户释放调整栏时发送WM_EXITSIZEMOVE消息
// 此处将根据新的窗口大小重置相关对象（如缓冲区、视图等）
case WM_EXITSIZEMOVE:
  mAppPaused = false;
  mResizing = false;
  mTimer.Start();
OnResize();
  return 0;
```

下面3个消息的处理过程比较简单，我们直接来看代码：

```
// 当窗口被销毁时发送WM_DESTROY消息
case WM_DESTROY:
  PostQuitMessage(0);
  return 0;

// 当某一菜单处于激活状态，而且用户按下的既不是助记键（mnemonic key）也不是加速
```

```
键
// （acceleratorkey）时，就发送WM_MENUCHAR消息
case WM_MENUCHAR:
  // 当按下组合键alt-enter时不发出beep蜂鸣声
  return MAKELRESULT(0, MNC_CLOSE);

// 捕获此消息以防窗口变得过小
case WM_GETMINMAXINFO:
  ((MINMAXINFO*)lParam)->ptMinTrackSize.x = 200;
  ((MINMAXINFO*)lParam)->ptMinTrackSize.y = 200;
  return 0;
```

最后，为了在代码中调用我们自己编写的鼠标输入虚函数，要按以下方式来处理与鼠标有关的消息：

```
case WM_LBUTTONDOWN:
case WM_MBUTTONDOWN:
case WM_RBUTTONDOWN:
  OnMouseDown(wParam, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
  return 0;
case WM_LBUTTONUP:
case WM_MBUTTONUP:
case WM_RBUTTONUP:
  OnMouseUp(wParam, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
  return 0;
case WM_MOUSEMOVE:
  OnMouseMove(wParam, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
  return 0;
```

为了使用GET_X_LPARAM和GET_Y_LPARAM两个宏，我们必须引入#include <Windowsx.h>。

4.5.6 初始化Direct3D演示程序

应用框架已经讨论完毕，现在就让我们用它来实现一个小程序。在这个程序中，我们基本不用去做什么，因为父类D3DApp几乎替我们完

成了所有的工作。在这里，我们的主要任务是从D3DApp中派生出自己的类，实现框架函数并在此为示例编写特定的代码。本书的所有程序都将遵循下面的模板。

```
#include "../../Common/d3dApp.h"
#include <DirectXColors.h>

using namespace DirectX;

class InitDirect3DApp : public D3DApp
{
public:
  InitDirect3DApp(HINSTANCE hInstance);
  ~InitDirect3DApp();

  virtual bool Initialize()override;

private:
  virtual void OnResize()override;
  virtual void Update(const GameTimer& gt)override;
  virtual void Draw(const GameTimer& gt)override;
};

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE prevInstance,
                         PSTR cmdLine, int showCmd)
{
  // 为调试版本开启运行时内存检测，方便监督内存泄露的情况
#if defined(DEBUG) | defined(_DEBUG)
  _CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
#endif

  try
  {
    InitDirect3DApp theApp(hInstance);
    if(!theApp.Initialize())
      return 0;

    return theApp.Run();
  }
  catch(DxException& e)
  {
    MessageBox(nullptr, e.ToString().c_str(), L"HR Failed", MB_OK);
    return 0;
  }
```

```
}

InitDirect3DApp::InitDirect3DApp(HINSTANCE hInstance)
: D3DApp(hInstance)
{
}

InitDirect3DApp::~InitDirect3DApp()
{
}

bool InitDirect3DApp::Initialize()
{
  if(!D3DApp::Initialize())
    return false;

  return true;
}

void InitDirect3DApp::OnResize()
{
  D3DApp::OnResize();
}

void InitDirect3DApp::Update(const GameTimer& gt)
{
}

void InitDirect3DApp::Draw(const GameTimer& gt)
{
  // 重复使用记录命令的相关内存
  // 只有当与GPU关联的命令列表执行完成时，我们才能将其重置
  ThrowIfFailed(mDirectCmdListAlloc->Reset());

  // 在通过ExecuteCommandList方法将某个命令列表加入命令队列后，我们便可以重置该命令列表。以
  // 此来复用命令列表及其内存
  ThrowIfFailed(mCommandList->Reset(
    mDirectCmdListAlloc.Get(), nullptr));

  // 对资源的状态进行转换，将资源从呈现状态转换为渲染目标状态
  mCommandList->ResourceBarrier(
    1, &CD3DX12_RESOURCE_BARRIER::Transition(
    CurrentBackBuffer(),
    D3D12_RESOURCE_STATE_PRESENT,
    D3D12_RESOURCE_STATE_RENDER_TARGET));

  // 设置视口和裁剪矩形。它们需要随着命令列表的重置而重置
```

```
  mCommandList->RSSetViewports(1, &mScreenViewport);
  mCommandList->RSSetScissorRects(1, &mScissorRect);

  // 清除后台缓冲区和深度缓冲区
  mCommandList->ClearRenderTargetView(
    CurrentBackBufferView(),
    Colors::LightSteelBlue, 0, nullptr);
  mCommandList->ClearDepthStencilView(
    DepthStencilView(), D3D12_CLEAR_FLAG_DEPTH |
    D3D12_CLEAR_FLAG_STENCIL, 1.0f, 0, 0, nullptr);

  // 指定将要渲染的缓冲区
  mCommandList->OMSetRenderTargets(1, &CurrentBackBufferView(),
    true, &DepthStencilView());

  // 再次对资源状态进行转换，将资源从渲染目标状态转换回呈现状态
  mCommandList->ResourceBarrier(
    1, &CD3DX12_RESOURCE_BARRIER::Transition(
    CurrentBackBuffer(),
    D3D12_RESOURCE_STATE_RENDER_TARGET,
    D3D12_RESOURCE_STATE_PRESENT));

  // 完成命令的记录
  ThrowIfFailed(mCommandList->Close());

  // 将待执行的命令列表加入命令队列
  ID3D12CommandList* cmdsLists[] = { mCommandList.Get() };
  mCommandQueue->ExecuteCommandLists(_countof(cmdsLists), cmdsLists);

  // 交换后台缓冲区和前台缓冲区
  ThrowIfFailed(mSwapChain->Present(0, 0));
  mCurrBackBuffer = (mCurrBackBuffer + 1) % SwapChainBufferCount;

  // 等待此帧的命令执行完毕。当前的实现没有什么效率，也过于简单
  // 我们在后面将重新组织渲染部分的代码，以免在每一帧都要等待
  FlushCommandQueue();
}
```

其中的一些方法我们还没有讨论过。ClearRenderTargetView方法会将指定的渲染目标清理为给定的颜色，ClearDepthStencilView方法则用于清理指定的深度/模板缓冲区。在每帧为了刷新场景而开始绘制之前，我们总是要清除后台缓冲区渲染目标和深度/模板缓冲区。

这两个方法的声明如下。

```
void ID3D12GraphicsCommandList::ClearRenderTargetView(
  D3D12_CPU_DESCRIPTOR_HANDLE RenderTargetView,
  const FLOAT ColorRGBA[ 4 ],
  UINT NumRects,
  const D3D12_RECT *pRects);
```

1. RenderTargetView：待清除的资源RTV。

2. ColorRGBA：定义即将为渲染目标填充的颜色。

3. NumRects：pRects数组中的元素数量。此值可以为0。

4. pRects：一个D3D12_RECT类型的数组，指定了渲染目标将要被清除的多个矩形区域。若设定此参数为nullptr，则表示清除整个渲染目标。

```
void ID3D12GraphicsCommandList::ClearDepthStencilView(
  D3D12_CPU_DESCRIPTOR_HANDLE DepthStencilView,
  D3D12_CLEAR_FLAGS ClearFlags,
  FLOAT Depth,
  UINT8 Stencil,
  UINT NumRects,
  const D3D12_RECT *pRects);
```

1. DepthStencilView：待清除的深度/模板缓冲区DSV。

2. ClearFlags：该标志用于指定即将清除的是深度缓冲区还是模板缓冲区。我们可以将此参数设置为D3D12_CLEAR_FLAG_DEPTH或D3D12_CLEAR_FLAG_STENCIL，也可以用按位或运算符连接两者，表示同时清除这两种缓冲区。

3. Depth：以此值来清除深度缓冲区。

4. Stencil：以此值来清除模板缓冲区。

5. NumRects：pRects数组内的元素数量。可以将此值设置为0。

6. pRects：一个D3D12_RECT类型的数组，用以指定资源视图将要被清除的多个矩形区域。将此值设置为nullptr，则表示清除整个渲染目标。

另一个新出现的方法是ID3D12GraphicsCommandList::OMSetRenderTargets，通过此方法即可设置我们希望在渲染流水线上使用的渲染目标和深度/模板缓冲区。（到目前为止，我们仅是把当前的后台缓冲区作为渲染目标，并只设置了一个主深度/模板缓冲区。但在本书的后续章节里，我们还将运用多渲染目标技术）。此方法的原型如下。

```
void ID3D12GraphicsCommandList::OMSetRenderTargets(
  UINT NumRenderTargetDescriptors,
  const D3D12_CPU_DESCRIPTOR_HANDLE *pRenderTargetDescriptors,
  BOOL RTsSingleHandleToDescriptorRange,
  const D3D12_CPU_DESCRIPTOR_HANDLE *pDepthStencilDescriptor);
```

1. NumRenderTargetDescriptors：待绑定的RTV数量，即pRenderTargetDescriptors数组中的元素个数。在使用多渲染目标这种高级技术时会涉及此参数。就目前来说，我们总是使用一个RTV。

2. pRenderTargetDescriptors：指向RTV数组的指针，用于指定我们希望绑定到渲染流水线上的渲染目标。

3. RTsSingleHandleToDescriptorRange：如果pRenderTargetDescriptors数组中的所有RTV对象在描述符堆中都是连续存放的，就将此值设为true，否则设为false。

4. pDepthStencilDescriptor：指向一个DSV的指针，用于指定我们希望绑定到渲染流水线上的深度/模板缓冲区。

最后要通过IDXGISwapChain::Present方法来交换前、后台缓冲区。与此同时，我们也必须对索引进行更新，使之一直指向交换后的当前后台缓冲区。这样一来，我们才可以正确地将下一帧场景渲染到新的后台缓冲区。

```
ThrowIfFailed(mSwapChain->Present(0, 0));
mCurrBackBuffer = (mCurrBackBuffer + 1) % SwapChainBufferCount;
```

图4.12所示的是本章例程的效果。



图4.12　本章例程的效果

4.6　调试Direct3D应用程序[46]

大多数的Direct3D函数会返回HRESULT错误码。我们的示例程序实则采用简单的错误处理机制来检测返回的HRESULT值。如果检测失败，则抛出异常，显示调用出错的错误码、函数名、文件名以及发生错误的行号。这些操作具体由d3dUtil.h中的代码实现：

```
class DxException
{
public:
  DxException() = default;
  DxException(HRESULT hr, const std::wstring& functionName,
    const std::wstring& filename, int lineNumber);

  std::wstring ToString()const;

  HRESULT ErrorCode = S_OK;
  std::wstring FunctionName;
  std::wstring Filename;
  int LineNumber = -1;
};

#ifndef ThrowIfFailed
#define ThrowIfFailed(x) \
{ \
  HRESULT hr__ = (x); \
  std::wstring wfn = AnsiToWString(__FILE__); \
  if(FAILED(hr__)) { throw DxException(hr__, L#x, wfn, __LINE__); } \
}
#endif
```

不难看出ThrowIfFailed必定是一个宏，而不是一个函数；若非如此，__FILE__和__LINE__将定位到ThrowIfFailed所在的文件与行，而非出错函数的文件与行。

L#x会将宏ThrowIfFailed的参数转换为Unicode字符串。这样一来，我们就能将函数调用所产生的错误信息输出到消息框当中。

对于Direct3D函数返回的HRESULT值，我们是这样使用宏对其进行检测的[47]：

```
ThrowIfFailed(md3dDevice->CreateCommittedResource(
  &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
  D3D12_HEAP_FLAG_NONE,
  &depthStencilDesc,
  D3D12_RESOURCE_STATE_COMMON,
  &optClear,
  IID_PPV_ARGS(mDepthStencilBuffer.GetAddressOf())));
```

整个程序逻辑都位于一个try/catch块之中：

```
  try
  {
    InitDirect3DApp theApp(hInstance);
    if(!theApp.Initialize())
      return 0;

    return theApp.Run();
  }
  catch(DxException& e)
  {
    MessageBox(nullptr, e.ToString().c_str(), L"HR Failed", MB_OK);
    return 0;
  }
```

如果返回的HRESULT是个错误值，则抛出异常，通过MessageBox函数输出相关信息，并退出程序。例如，在向CreateCommittedResource方法传递了一个无效参数时，我们便会看到图4.13所示的消息框。



图4.13　当返回的HRESULT是个错误码时，会弹出消息框并显示类似的错误信息

4.7　小结

1．可以把Direct3D 看作是一种介于程序员和图形硬件之间的“桥梁”。借此，程序员便可以通过调用Direct3D函数来实现把资源视图绑定到硬件渲染流水线、配置渲染流水线的输出以及绘制3D几何体等操作。

2．组件对象模型（COM）是一种可以令DirectX不依赖于特定语言且向后兼容的技术。Direct3D程序员不需知道COM的具体实现细节，也无需了解其工作原理，只需知晓如何获取和释放COM接口即可。

3．1D、2D、3D纹理分别类似于由数据元素所构成的1D、2D、3D数组。纹理元素的格式必定为`DXGI_FORMAT`枚举类型中的成员之一。除了常见的图像数据，纹理也能存储像深度信息等其他类型的数据（如深度缓冲区就是一种存储深度值的纹理）。GPU可以对纹理进行特殊的操作，比如运用过滤器和进行多重采样。

4．为了避免动画中发生闪烁的问题，最好将动画帧完全绘制到一种称为后台缓冲区的离屏纹理中。只要依此行事，显示在屏幕上的就会是一个完整的动画帧，观者也就不会察觉到帧的绘制过程。当动画帧被绘制在后台缓冲区后，前台缓冲区与后台缓冲区的角色也就该互换了：为了显示下一帧动画，此前的后台缓冲区将变为前台缓冲区，而此前的前台缓冲区亦会变成后台缓冲区。后台和前台缓冲区交换角色的行为称为呈现（present）。前台和后台缓冲区构成了交换链，在代码中通过

IDXGISwapChain接口来表示。使用两个缓冲区（前台和后台）的情况称作双缓冲。

5．假设场景中有一些不透明的物体，那么离摄像机最近的物体上的点便会遮挡住它后面一切物体上的对应点。深度缓冲就是一种用于确定在场景中离摄像机最近点的技术。通过这种技术，我们就不必再担心场景中物体的绘制顺序了。

6．在Direct3D中，资源不能直接与渲染流水线相绑定。为此，我们需要为绘制调用时所引用的资源指定描述符。我们可将描述符对象看作是GPU识别以及描述资源的一种轻量级结构体。而且，我们还可以为同一种资源创建不同的描述符。如此一来，一种资源就可以具有多种用途。例如，我们可以借此将同一种资源绑定到渲染流水线的不同阶段，或者用不同的DXGI_FORMAT成员将它描述为不同的格式。应用程序可通过创建描述符堆来为描述符分配所需的内存。

7．ID3D12Device是Direct3D中最重要的接口，我们可以把它看作是图形硬件设备的软件控制器。我们能够通过它来创建GPU资源以及其他用于控制图形硬件的特定接口。

8．每个GPU中都至少有一个命令队列。CPU可通过Direct3D API用命令列表向该队列提交命令，而这些命令则指挥GPU执行某些操作。在命令没有到达队列首部以前，用户所提交的命令是无法被执行的。如果命令队列内为空，则GPU会因为没有任务要去处理而处于空闲状态；但若命令队列被装得太满，则CPU将在某个时刻因提交命令的速度追上

GPU执行命令的速度而进入空闲状态。值得一提的是，这两种情景其实都没有充分地利用系统资源。

9．GPU是系统中与CPU一起并行工作的第二种处理器。有时，我们需要对CPU与GPU进行同步。例如，若GPU命令队列中有一条引用某资源的命令，那么在GPU完成此命令的处理之前，CPU就不能修改或销毁这一资源。任何同步方法都会导致其中的一种处理器处于一段等待和空闲的状态，这意味着两种处理器并没有被充分利用，因此，我们应尽量减少同步的次数，并缩短同步的时间。

10．性能计数器是一种高精度的计时器，它是测量微小时间差的一种有效工具。例如，我们可以用它来测量两帧之间的间隔时间。性能计时器使用的时间单位称为计数（count）。`QueryPerformanceFrequency`函数输出的是性能计时器每秒的计数，可用它将计数单位转换为秒。性能计时器的当前时间值（以计数为单位测量）可用`QueryPerformanceCounter`函数获得。

11．通过统计时间段Δt内处理的帧数即可计算出每秒的平均帧数（FPS）。设n为时间Δt内处理的帧数，那么该时间段内每秒的平均帧数为$FPS_{avg} = n/\Delta t$。采用帧率进行考量可能会对性能造成一些误判，相对而言，“处理一帧所花费时间”这个统计信息可能更加精准、直观。以秒为单位表示的每帧平均处理时间可以用帧率的倒数来计算，即$1/FPS_{avg}$。

12．示例框架为本书的全部例程都提供了统一的接口。d3dUtil.h、

d3dUtil.cpp、d3dApp.h和d3dApp.cpp文件封装了所有应用程序必须实现的标准初始化代码。封装这些代码便隐藏了相关的细节，这样的话，我们就可以将精力集中在不同示例的特定主题之上。

13．为了开启调试模式需要启用调试层（`debugController->EnableDebugLayer()`）。如此一来，Direct3D就会把调试信息发往VC++的输出窗口。

---

[1] 即render target，简单来讲，渲染目标是为了渲染场景而将像素绘制到的特定缓冲区（buffer）。通常是占用部分显存的后台缓冲区，以及纹理（详见后文）。

[2] 在预览版中，此函数为`ID3D12CommandList::ClearRenderTargetView`。在正式版中，预览版`ID3D12CommandList`接口下的函数基本移至`ID3D12GraphicsCommandList`接口下。

[3] pixel，构成图像的基本元素。从图形角度来讲，可认为像素是一种图像的采样单位（将图像以像素为基础进行划分，再于像素中进行采样）。因此，两张同样大小的图片，分辨率高者，意味着像素数量越多，细节越丰富，画面就越清晰。由于实际显示上的原因（后面注释会提到），也赋予了像素“大小”的概念。在Direct3D中，像素被抽象为具有一定长宽的色块。

[4] 关于mipmap的相关知识可参见9.5.2节。

[5]　屏幕等显示设备采用的是rgb色彩空间，为了区分美术中的传统三原色（红黄蓝）在此称红绿蓝为三基色。若要了解其他色彩空间，请参考其他文献。

[6]　原文为reinterpret cast，其实在这里翻译为重新解释数据类型更妥。但考虑到C++语言术语予以保留。

[7]　感兴趣的读者可以查看“三重缓冲”与“垂直同步”的有关知识。

[8]　至于二者是否需要区分，具体还要看应用场景。比如谈到像素与纹素的映射关系时，必须将这两个概念予以区分。文中谈到的基本上是约定俗成的叫法。

[9]　即sampling，也作取样，本是信号处理方面的术语。在本书中，可认为该操作是以特定的模式，从连续的图像数据中采集出离散的关键颜色信息。

[10]　即不同厂商所生产的显示器（甚至同一厂商不同型号的显示器）屏幕中的像素都会有特定的形状、大小以及数量，这取决于其子像素（subpixel）的形状、大小乃至排列顺序。因此，导致直线实际上是由数量有限、离散而非连续的“点”构成，继而造成其“阶梯”状的锯齿效应。注意，在这段注解里所说的硬件显示设备上的“像素”，有面积、形状，而“子像素”则是实现硬件像素三基色通道的基本单位，即一个“子像素”表达一个颜色通道。请勿与正文中的术语混淆。

[11]　前文注释中曾提到，其实是提升了单位面积上的像素数量，从

而“缩小”了像素的大小。

[12] 文中出现的“子像素（subpixel）”是指在像素内部再次细分出的小像素，用以采样。

[13] 除了文中这两种常见的反锯齿（anti-aliasing）手段，还有许多以不同方式实现的反锯齿技术，有兴趣的读者可以自行研究。

[14] 若是黑白印刷，则文中的“绿色”即深灰色，“浅绿色”为浅灰色，白色则表示该像素保持之前的颜色。

[15] Windows 10在1607版、1703版与1709版分别引入了`ID3D12Device`接口的新版本`ID3D12Device1`、`ID3D12Device2`与`ID3D12Device3`。官方文档建议用户，在1607至1703版本的操作系统上采用`ID3D12Device1`接口，在1703至1709版本的操作系统上采用`ID3D12Device2`接口，在`1709`及其后续版本的操作系统上使用`ID3D12Device3`接口（虽然这样做的人并不多，甚至包括官方示例。想用新功能的读者倒可以试一试）。查阅对应文档便可知每次向`ID3D12Device`接口添加的新功能。

[16] 作者在前文也曾写到，在写作此书时采用的是预览版SDK，但它与正式版SDK中的少量API以及枚举项却有细微差别。微软官方现在对早期文档都标有“preliminary”字样，但在撰写此脚注的时候有些正式版文档的个别细节仍未完全修正。在此，将文中预览版的API均修正为当前的正式版，并在第一次出现时以注解形式给出预览版的原API，算是对DirectX 12历史的见证吧~代码中的枚举类型

D3D12_MULTISAMPLE_QUALITY_LEVEL_FLAGS在预览版中为D3D12_MULTISAMPLE_QUALITY_LEVELS_FLAG。

[17] 此处描述不准确。Direct3D 12并不支持创建MSAA交换链！

[18] 同理，当前Direct3D 12的对应版本有D3D_FEATURE_LEVEL_12_0、D3D_FEATURE_LEVEL_12_1。

[19] 扫描式显示设备的工作方式有两种：逐行扫描与隔行扫描。试想，将显示设备的屏幕划分为多个行，称之为“场（field）”，奇数行称为奇数场（upper field），偶数行则称为偶数场（lower field）。顾名思义，逐行扫描即在显示每一帧画面时都从上至下逐个场连续扫描。但根据人眼的视觉暂留效应，便可以每帧仅扫描一种场，交替扫描。

[20] GPU memory，也有直译作GPU内存等。显卡通常是一块带有PCIe总线接口的物理电路（这里仅谈独立显卡），GPU较之于显卡的地位大致相当于CPU较之于主板。相应的，GPU控制的显存基本相当于CPU控制的内存，而后者在本书中也常被称为系统内存（system memory）。CPU内部有多级缓存与寄存器，分别用于缓存指令与控制CPU；GPU内部亦有缓存与寄存器，分别用于缓存纹理、缓存着色器指令等以及控制GPU。有的文献在划分GPU的组成结构时，会把GPU的寄存器及其控制的内存统称为GPU memory （GPU存储器）。

[21] 像书中给出的这类地址，完全可以通过搜索神秘代码上车，比如这里的mt186622。

[22]　相对于Direct3D 12而言，Direct3D 11支持两种绘制方式：即立即渲染（immediate rendering，利用immediate context实现）以及延迟渲染（deferred rendering，利用deferred context实现）。前者将缓冲区中的命令直接借驱动层发往GPU执行，后者则与本文中介绍的命令列表模型相似（但执行命令列表时仍然要依赖immediate context）。前者延续了Direct3D 11之前一贯的绘制方式，而后者则为Direct3D 11中新添加的绘制方式。到了Direct3D 12便取消了立即渲染方式，完全采用“命令列表->命令队列”模型，使多个命令列表同时记录命令，借此充分发挥多核心处理器的性能。可见，Direct3D 11在绘制方面乃承上启下之势，而Direct3D 12则进行了彻底的革新。

[23]　Windows 10在1703版与1709版分别引入了`ID3D12GraphicsCommandList`接口的新版本`ID3D12Graphics-CommandList1`与`ID3D12GraphicsCommandList2`。官方文档建议用户在1703至1709版本的操作系统上采用`ID3D12GraphicsCommandList1`接口，在`1709`及其后续版本的操作系统上使用`ID3D12GraphicsCommandList2`接口（虽然这么做的人不多，甚至包括官方示例。但想用新功能的读者倒可以试一试）。查阅对应文档便可了解每次向该接口添加的新功能。

[24]　除了本书文中介绍的这两种命令列表类型之外，还有如`D3D12_COMMAND_LIST_TYPE_COMPUTE`（仅接收与通用计算有关的命令）以及`D3D12_COMMAND_LIST_TYPE_COPY`（只接收与复制操作相关的命令）等类型。

[25]　bundle是种二级命令列表，可将它看作是一组状态和命令的集合，把它多次挂靠在命令列表上即可对其进行复用。

[26]　Windows 10在1709版引入了`ID3D12Fence`接口的新版本`ID3D12Fence1`。官方文档建议用户在自1709版本的操作系统上采用`ID3D12Fence1`接口。查阅对应文档便可了解向该接口添加的新功能。读者可对此加以尝试。

[27]　`ID3D12CommandQueue::Signal`方法从GPU端设置围栏值，而`ID3D12Fence::Signal`方法则从CPU端设置围栏值。

[28]　在Direct3D 11中，这些工作全权交由驱动来管理，因此性能会稍差。（在Direct3D 12中资源状态靠手动进行转换。就此而言，应该就不需要驱动层介入资源状态的跟踪了。但是作者说“仍有一个资源状态跟踪系统”，查询之下似乎只有调试层（debug layer）才有这个用来查错的“追踪系统”，文中说法有待考证）。

[29]　预览版里`D3D12_RESOURCE_BARRIER`结构体的名称为`D3D12_RESOURCE_BARRIER_DESC`。

[30]　已更名为D3D12Multithreading，可在微软官方示例中找到。

[31]　简单来讲，在操作系统中的显卡不能发挥效用等情况（详见文档），WARP便会挺身而出。可将它当作一个不依赖于任何硬件图形适配器的纯软件渲染器。根据wikipeida给出的信息，Windows 10上的WARP版本可支持的最高功能级别为feature level 12_1。但到现在为止，

微软的官方网站并未更新WARP的相关文档。

[32]　前文注释曾提到：Direct3D 12并不支持创建MSAA交换链，因此也就不能在运行时改动交换链的MSAA参数！

[33]　伪代码：目标描述符句柄=
GetCPUDescriptorHandleForHeapStart() + mCurrBackBuffer *
mRtvDescriptorSize。

[34]　ID3D12Resource接口将物理内存与堆资源抽象组织为可处理的数据数组与多维数据，从而使CPU与GPU可以对这些资源进行读写。

[35]　在预览版里此结构体中D3D12_RESOURCE_FLAGS名为D3D12_RESOURCE_MISC_FLAG。

[36]　在早期版本里此项似乎多次更名，如D3D12_RESOURCE_MISC_ALLOW_DEPTH_STENCIL、D3D12_RESOURCE_M
有笔误的可能）。

[37]　预览版中D3D12_CPU_PAGE_PROPERTY的名称为D3D12_CPU_PAGE_PROPERTIES。

[38]　在预览版中，结构体D3D12_HEAP_FLAGS及其成员D3D12_HEAP_FLAG_NONE的名称分别为D3D12_HEAP_MISC_FLAG与D3D12_HEAP_MISC_NONE。

[39]　在预览版中，结构体D3D12_RESOURCE_STATES名

为D3D12_RESOURCE_USAGE，而深度/模板缓冲区的状态也不像当前分为_READ与_WRITE读写两种，仅为D3D12_RESOURCE_USAGE_DEPTH一种状态。至于资源初始状态也不是D3D12_RESOURCE_STATE_COMMON，而是D3D12_RESOURCE_USAGE_INITIAL。

[40]　从字面意思上可得出视口坐标所采用的坐标系，即以缓冲区左上角为原点，x、y轴的正方向分别为水平向右与垂直向下。另外视口坐标最小值D3D12_VIEWPORT_BOUNDS_MIN为-32768，不妨取负值试一试效果如何。

[41]　根据文档来看，Direct3D 12内封装了一组对应的API，见《Timing》（dn903946）。

[42]　这段对白在当前的文档中已经看不到了，可以从别处的文献中看到蛛丝马迹。原文为："On a multiprocessor computer, it should not matter which processor is called. However, you can get different results on different processors due to bugs in the basic input/output system (BIOS) or the hardware abstraction layer (HAL)."

[43]　这段代码注释得比较混乱，其中的停止（stop）与暂停（pause）意义相同。

[44]　参见本书附录A。

[45]　个人感觉除了"宽高比"这一译法都有误导嫌疑，即在大多数场合都应为"横向尺寸/纵向尺寸"。

[46]　PIX重现江湖。详见《Introducing PIX on Windows (beta)》一文及其后续博客文章。另外，较新版本的visual studio已经集成了较强大的图形诊断调试工具，详见《调试应用程序（Debugging Applications）》（mt243869）下的《调试GPU代码（Debugging GPU Code）》（hh873126）与《图形诊断（调试DirectX图形）（Graphics Diagnostics (Debugging DirectX Graphics)）》（hh315751）。玩一玩这几样工具会对渲染流水线有直观印象。另有gpuview，借此可观察到GPU与CPU之间的通信过程，乃至驱动向GPU硬件队列发送命令等工作流，加深同步方面的理解。

[47]　在预览版中，`CD3DX12_HEAP_PROPERTIES`名为`CD3D12_HEAP_PROPERTIES`，而且DirectX 12辅助结构体的前缀大多已由`CD3D12`变更为`CD3DX12`。由于Windows 10的更新策略相对于以往有了改变，因此读者也应当注意具体系统版本对DirectX 12的更新（New Releases，mt748631）。

第5章　渲染流水线

本章要探讨的主题是渲染流水线[1]（rendering pipeline）。如果给出一台具有确定位置和朝向的虚拟摄像机（virtual camera）以及某个3D场景的几何描述，那么渲染流水线则是以此虚拟摄像机为视角进行观察，并据此生成给定3D场景2D图像的一整套处理步骤（见图5.1）。本章的内容更偏于理论——而下一章会用Direct3D将理论付诸实践。在讲解渲染流水线之前，先要花费些时间解决两个问题：首先要讨论的是3D视觉要素（即通过扁平的2D显示器屏幕却能观察到3D立体场景的视觉因素），其次是解释如何用数学方法来表示以及使用颜色，并在Direct3D代码中加以实现。

学习目标：

1．了解用于在2D图像中表现出场景立体感和空间深度感等真实效果的关键因素。

2．探索如何用Direct3D表示3D对象。

3．学习怎样建立虚拟摄像机。

4．理解渲染流水线——根据给定3D场景的几何描述，生成其2D图像的流程。

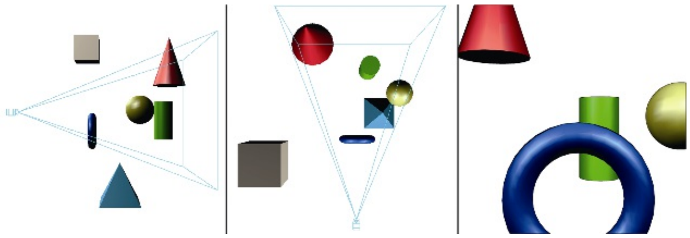


图5.1　左侧的图例展示了一台具有特定位置和朝向的虚拟摄像机，以及置于3D场景中一些物体的侧 视图。中间的图例演示了同样的场景，但采用的是由上至下的俯视视角。其中的“四棱锥”圈定的 是观察者通过摄像机所看到的空间范围，而位于此四棱锥之外的物体（或部分物体）则是观察者 所看不到的。右侧的图例展示即为观察者通过摄像机所看到的2D图像

5.1　3D视觉即错觉？

在开启3D计算机图形学的旅程之前，还有一个尚待解决的简单问题摆在我们眼前：应怎样将3D场景的空间深度感和立体感在2D平面显示器的屏幕中表现出来呢？幸运的是，前人早已深入地研究过这个问题了，毕竟艺术家们在2D画布上创作3D艺术作品的历史已持续了几个世纪。在本节中，我们将探讨几种使图像在2D平面上“立体”起来的关键技巧。

假设我们看到一段笔直的铁轨，它向远方绵延，望不到尽头。铁路的双轨虽然互相平行，但是，当我们站在铁路中央向远处张望时就会发现：两条铁轨随着距离的增加在逐渐靠近，甚至在无限远处相交了。这其实是人类视觉系统的一个特性：即从观察效果上看，平行线最终会相交于消失点（vanishing point，又称灭点），如图5.2所示。

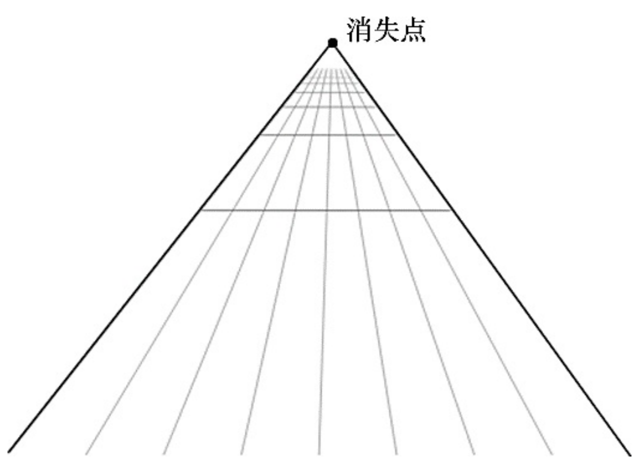


图5.2　平行线在视觉上终会汇于消失点。
艺术家常称之为线性透视（linear perspective， 亦有译作线条透视）

通过对人眼观察物体的过程进行简单的考察不难得出以下结论：随着（z方向）深度的增加，物体会显得越来越小；这也就是我们常说的“近大远小”。例如，远山上的房子看起来很小，而离我们较近的小树相对而言看起来反而较大。图5.3展现了这样一幅简易的场景：两排平行的石柱相对而立。石柱的尺寸本完全相同，但随着向远处推进，它们看起来一个比一个小。我们还可以发现，这两排石柱最后也将相交于地平线上的消失点。

我们都知道物体重叠（object overlap）的概念，即不透明物体能够遮挡住其后侧物体的局部（或整体），如图5.4所示。这是一个重要的概念，它传达了不同物体在场景中的深度顺序关系。而我们在第4章中也已经讨论过如何在Direct3D中借助深度缓冲区来确定那些应当受到遮蔽而不是绘制出来的像素。



图5.3　图中所有的石柱本大小相同，但观察者看到的效果却是石柱随距离的增加而逐渐变小



图5.4　一组由于前后位置关系而彼此遮挡的物体（这便是具有重叠关系的物体）

我们把目光再转向图5.5。图的左侧是一个不受光照的球体，而右侧是一个被光照射的球体。我们能明显地察觉到：左边的球体看起来颇为平坦，丝毫没有立体感——可能它甚至都不是球体，而仅是一个2D圆片而已！所以，光照和阴影的处理在刻画3D物体的实体[2]形状和立体感中扮演着至关重要的角色。

最后，图5.6所示的是一艘宇宙飞船及其阴影。阴影担负着两个关键的任务。首先，它暗示了光源在场景中的相对位置。其次，它反映了飞船起飞的大致高度。

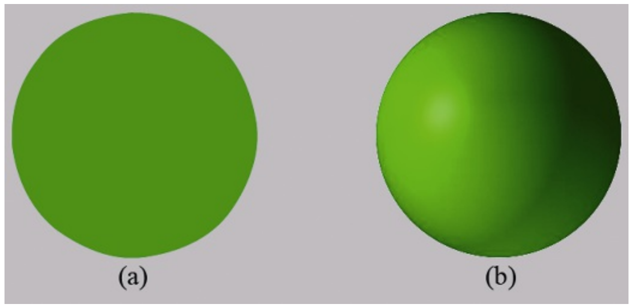


图5.5　光照对球体立体感效果的影响

（a）不受光照的球体看上去就是个2D图形

（b）受光照射的球体颇具3D立体感



图5.6　一艘宇宙飞船及其阴影。阴影不仅暗示着场景中的光源位置，而且反映了飞船离地高度这一信息

刚刚讨论过的种种观察结论都是我们日复一日积累下来的直观经验，因此是毋庸置疑的。尽管如此，把这些司空见惯的规律总结出来，并且牢记于心，对我们在3D计算机图形学上的学习和工作仍都是大有裨益的。

5.2 模型的表示

实际上，实体3D对象是借助三角形网格（triangle mesh）来近似表示的，因而我们要以三角形作为3D物体建模的基石。如图5.7所示，我们能用三角形网格近似地模拟出任何真实世界中的3D物体。通常来讲，模拟一个物体所用的三角形越多，那么模型就与目标物体越接近，这是因为模型会随之获得更为丰富的细节。当然，建模所用的三角形越多，也就需要更强大的计算处理能力，所以要根据应用受众的硬件性能做出权衡。除了三角形，点和线也有其用武之地。例如，我们可以利用一系列宽度为1像素的短线段绘制出一条近似曲线。

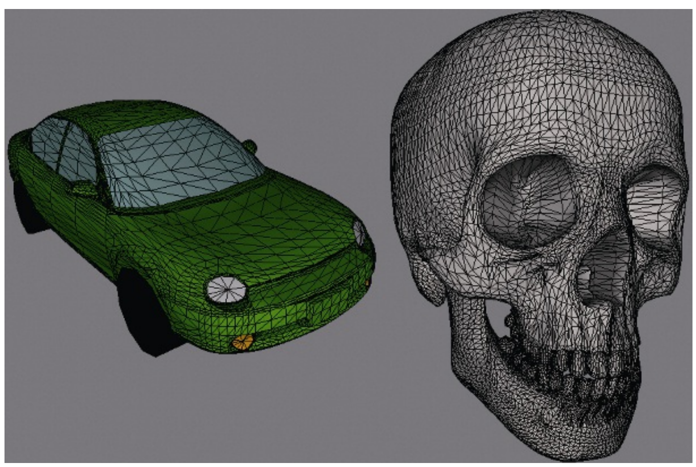


图5.7　一辆用三角形网格近似模拟出来的小 汽车和一颗由三角形网格近似表示的骷髅头

看到图5.7中使用的大量三角形，又一个问题逐渐浮出水面：若要手动列出这些三角形来模拟3D物体，实在是一件太麻烦的事儿了。除了最简单的模型，我们可以使用一种叫作3D建模工具（3D modeler）的专用软件，来生成和处理复杂的3D对象。在这些建模软件的可视化交

互环境里，用户可以运用其中丰富的工具集构建出复杂而富有真实感的网格，因此，这种软件使整个建模的流程要方便快捷得多。在游戏开发方面，流行的建模软件有3D Studio Max、LightWave 3D、Maya、Softimage|XSI[3]和Blender。其中，Blender是开源和免费爱好者的福音。然而，在本书的第一部分中，我们仍将采用手动或运用数学公式的方式来生成3D对象（比如，通过参数方程就可以方便地生成用于模拟圆柱和球体的三角形列表），在第三部分中，我们将展示如何加载和显示以3D建模程序生成的3D模型。

5.3　计算机色彩基础

计算机显示器中的每个像素发出的都是红、绿、蓝三色混合光[4]。当混合的光线进入观察者眼中，照射到视网膜的特定区域时，视锥细胞便受到刺激产生神经冲动，并通过视神经传至大脑，大脑继而解释传来的信号并感知颜色。因为混合光的变化各异，细胞受到的刺激也各不相同，所以我们就能感知到颜色间的差异了。图5.8上图给出了红、绿、蓝三基色混合成不同颜色的示例，下图展示的则是不同强度的红色。通过将这3种颜色分量按不同强度的混合在一起，我们便能描述出显示真实感图像所需的一切颜色。

上至Adobe Photoshop等各种绘图程序，下至Win32中的`ChooseColor`（选择颜色）对话框（见图5.9）都可以作为初学者以RGB（red，green，blue，红绿蓝）值来描述颜色的练习途径。在此，我们可通过尝试不同的RGB组合来观察它们所混合出的各种颜色。

便于描述光的强度，我们常将它量化为范围在0～1归一化区间中的值。

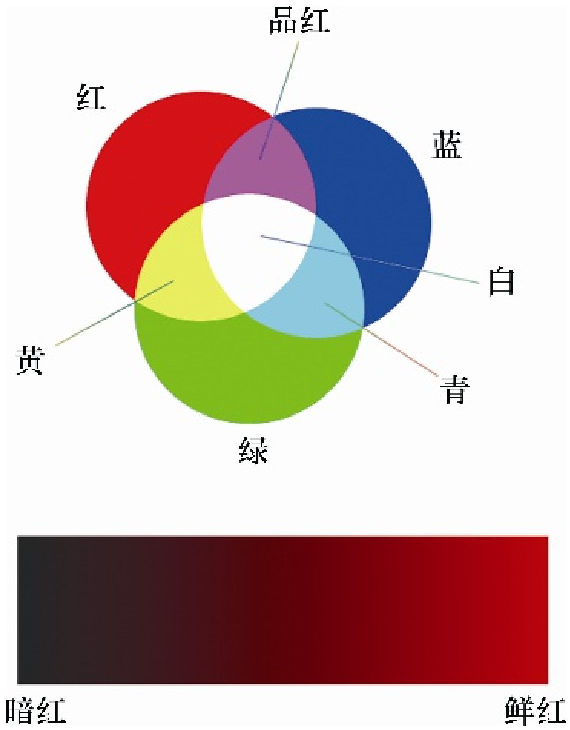


图5.8　（上图）红、绿、蓝3种纯色混合所得到的新颜色。
（下图）控制红光的强度便可调配出不同明度的红色

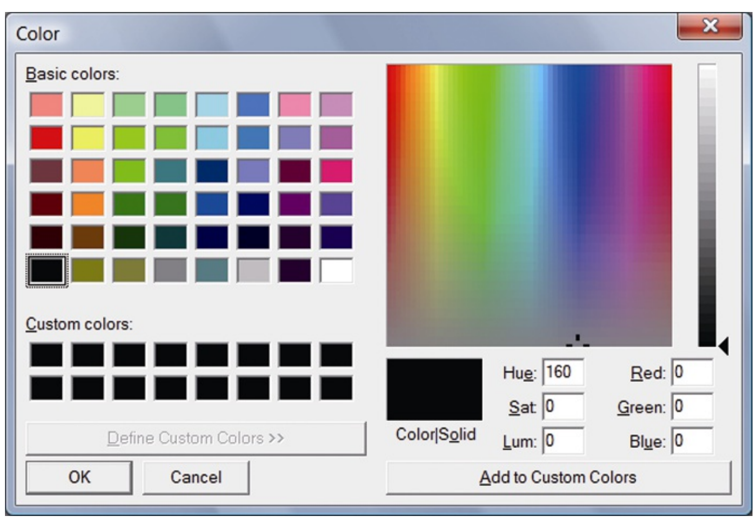


图5.9　ChooseColor对话框

每款显示器所能发出的红、绿、蓝三色光的强度都是有限的。为了便于描述光的强度，我们常将它量化为范围在0～1归一化区间中的值。

0代表无强度，1则表示强度最大，处于两者之间的值就表示对应的中间强度。例如，强度值（0.25, 0.67, 1.0）就表明此光线由强度为25%的红色光、强度为67%的绿色光以及强度为100%的蓝色光混合而成。由此例可以看出，我们能用3D向量（r, g, b）来表示颜色，其中$0 \leqslant r, g, b \leqslant 1$，这3种颜色分量分别代表红、绿、蓝三色光在混合光中的强度。

5.3.1　颜色运算

向量的部分运算规则在颜色向量上同样适用。例如，我们可以使两个颜色向量相加来得到新的颜色：

$$(0.0, 0.5, 0) + (0, 0.0, 0.25) = (0.0, 0.5, 0.25)$$

这就是说，通过混合中等强度的绿色和低等强度的蓝色，便会得到深绿色。

也可以通过颜色向量之间的减法运算来获得新的颜色：

$$(1, 1, 1) - (1, 1, 0) = (0, 0, 1)$$

由上式可以看出，从白色中去掉红色和绿色的成分，便可得到蓝色。

标量乘法也是有效的，请考虑下式：

$$0.5(1, 1, 1) = (0.5, 0.5, 0.5)$$

此式将白色的各颜色分量取半，继而得到中等强度的灰色。另外，可通过2(0.25, 0, 0) = (0.5, 0, 0)运算将红色分量的强度加倍。

显而易见的是，像点积和叉积这样的运算法则就不适用于颜色向量了。不过，颜色向量也有它们自己专属的颜色运算，即分量式（modulation或componentwise）乘法。它的定义为：

$$(c_r, c_g, c_b) \otimes (k_r, k_g, k_b) = (c_r k_r, c_g k_g, c_b k_b)$$

这种运算主要应用于光照方程。例如，假设有颜色为（r, g, b）的入射光线，照射到一个反射50%红色光、75%绿色光、25%蓝色光且吸收剩余光的表面。那么，我们就可以据此给出反射光线的颜色：

$$(r, g, b) \otimes (0.5,\ 0.75,\ 0.25) = (0.5r, 0.75g, 0.25b)$$

通过此式即可看出，由于此表面会吸收一部分入射光，所以当它照射在该平面上时会损失掉部分颜色强度。

在进行颜色运算的过程中，颜色分量有可能会超出[0, 1]这个区间。如，思考(1, 0.1, 0.6) + (0, 0.3, 0.5) = (1, 0.4, 1.1)这个等式。由于1.0代表颜色分量的最大强度，所以任何光的强度都不能超过此值。因此，我们就只得将值为1.1的强度与1.0这一上限强度视作等同，将1.1钳制（clamp）为1.0。同样地，显示器也不能发出强度为负值的光，所以亦应把负的颜色分量（由减法运算所得到的结果）钳制为0.0。

5.3.2　128位颜色

事实上，我们通常还会用到另一种名为**alpha**分量（alpha component）的颜色分量。alpha分量常用于表示颜色的不透明度[5]（opacity。值为0.0表示完全透明，值为1.0表示不透明），它在混合（blending）技术（第10章）中起到了至关重要的作用（因为我们目前还用不到混合技术，所以暂将alpha分量置为1）。这就是说，把alpha分量算在内的话，我们就可以用4D向量(r, g, b, a)来表示每一种颜色，分量需要满足$0 \leqslant r, g, b, a \leqslant 1$。为了用128位（128bit）数据来表示一种颜色，每个分量都要使用浮点值。由于每种颜色刚好能用数学上的4D向量来表示，所以我们也就能在代码中用`XMVECTOR`类型来描述它们。在通过DirectXMath向量函数来进行颜色运算（如颜色的加法运算、减法运算和标量乘法运算）的同时，我们也能借助SIMD技术加快数据的处理速度。DirectXMath库针对分量式乘法运算提供了下列函数：

```
XMVECTOR XM_CALLCONV XMColorModulate(  // 返回c1⊗c2
    FXMVECTOR C1,
  FXMVECTOR C2);
```

5.3.3　32位颜色

为了用32位（32bit）数据表示一种颜色，每个分量仅能分配到1个字节。因此，每个占用8位字节的颜色分量就可以分别描述256种不同的颜色强度——0代表无强度，255是最大强度，处于两者之间的值也就表示相应的中间强度。每种颜色分量占用空间虽然看起来很小，但是它们的全部组合（$256 \times 256 \times 256 = 16\ 777\ 216$）却能表示出千万种不同的颜色。DirectXMath库（`#include <DirectXPackedVector.h>`）

在DirectX::PackedVector命名空间中提供了下面的结构用于存储32位颜色[6]：

```
namespace DirectX
{
namespace PackedVector
{
// ARGB颜色表示法;以8-8-8-8位的无符号归一化整数分量封装为一个32位的整数
// 将alpha、红、绿、蓝4种分量分别用8位无符号归一化整数表示，以此封装32位归一化颜色
// alpha分量存于最高8位有效位，而蓝色分量则存于最低8位有效位(A8R8G8B8)
// [32] aaaaaaaa rrrrrrrr gggggggg bbbbbbbb [0][7]
struct XMCOLOR
{
  union
  {
    struct
    {
      uint8_t b; // Blue: 0/255 to 255/255
      uint8_t g; // Green: 0/255 to 255/255
      uint8_t r; // Red: 0/255 to 255/255
      uint8_t a; // Alpha: 0/255 to 255/255
    };
    uint32_t c;
  };

  XMCOLOR() {}
  XMCOLOR(uint32_t Color) : c(Color) {}
  XMCOLOR(float _r, float _g, float _b, float _a);
  explicit XMCOLOR(_In_reads_(4) const float *pArray);

  operator uint32_t () const { return c; }

  XMCOLOR& operator= (const XMCOLOR& Color) { c = Color.c; return
    *this; }
  XMCOLOR& operator= (const uint32_t Color) { c = Color; return *this;
    }
};
} // PackedVector 命名空间结束
} // DirectX 命名空间结束
```

通过将整数范围[0, 255]映射到实数区间[0, 1]，就可以将32位颜色

转换为128位颜色，具体做法是将每个分量分别除以255。也就是说，如果有整数$0 \leqslant n \leqslant 255$，那么$0 \leqslant \frac{n}{255} \leqslant 1$即为归一化范围0～1的颜色强度。例如，设有一32位颜色(80, 140, 200, 255)，将其转换成对应的128位颜色的过程为：

$$(80,\ 140,\ 200,\ 255) \rightarrow \left(\frac{80}{255}\frac{140}{255}\frac{200}{255}\frac{255}{255}\right) \approx (0.31,\ 0.55,\ 0.78,\ 1.0)$$

相反地，128位颜色也可以转换为32位颜色，方法是将每个颜色向量分别乘以255，再四舍五入取整，如：

$$(0.3, 0.6,\ 0.9,\ 1.0) \rightarrow (0.3 \times 255, 0.6 \times 255, 0.9 \times 255, 1.0 \times 255) = (77,\ 153,\ 230,\ 255)$$

由于在XMCOLOR中通常将4个8位颜色分量封装为一个32位整数值（例如，一个unsigned int类型的值），因此在32位颜色与128位颜色互相转换的过程中常常需要进行一些额外的位运算（提取出每个分量）。对此，DirectXMath库中定义了一个获取XMCOLOR类型实例并返回其相应XMVECTOR类型值的函数：

```
XMVECTOR XM_CALLCONV PackedVector::XMLoadColor(
  const XMCOLOR* pSource);
```

图5.10展示了将4个8位颜色分量封装为UINT（32位无符号整数）类型的具体细节。注意，这仅是封装颜色分量的方式之一。除了ARGB之外，还有ABGR以及RGBA这两种格式，只不过XMCOLOR类中使用的格式为ARGB而已。另外，DirectXMath库还提供了一个可将XMVECTOR转换至XMCOLOR的函数：

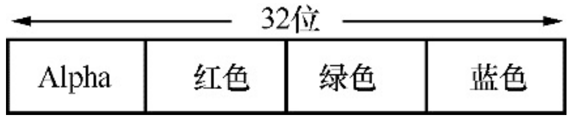


图5.10　32位颜色表示法，为alpha、红、绿、蓝4种分量都各分配了1字节

```
void XM_CALLCONV PackedVector::XMStoreColor(
  XMCOLOR* pDestination,
  FXMVECTOR V);
```

一般来说，128位颜色值常用于高精度的颜色运算（例如位于像素着色器中的各种运算）。在这种情况下，由于运算所用的精度较高，因此可有效降低计算过程中所产生的误差。但是，最终存储在后台缓冲区中的像素颜色数据，却往往都是以32位颜色值来表示。而目前的物理显示设备仍不足以充分发挥出更高色彩分辨率的优势[Verth04]。

5.4　渲染流水线概述

若给出某个3D场景的几何描述，并在其中架设一台具有确定位置和朝向的虚拟摄像机，那么渲染流水线（rendering pipeline）是以此摄像机为观察视角而生成2D图像的一系列完整步骤。图5.11左侧展示的是组成渲染流水线的所有阶段，而右侧则是显存资源。从资源内存池指向渲染流水线阶段的箭头，表示该阶段可以访问资源并以此作为输入。例如，在像素着色器阶段（pixel shader stage），渲染流水线为了完成用户分派的任务，可以从显存所存的纹理资源中读取数据。从渲染流水线阶段指向内存的箭头，则意味着该阶段可以向GPU资源写入数据。例如，在输出合并（器）阶段（output merger stage）把数据写到像后台缓冲区和深度/模板缓冲区这样的纹理之中。同时也可以看到，位于输出合并阶段的箭头是双向的（说明此阶段可读写GPU资源）。如我们所见，大多数阶段都是不能向GPU资源进行写操作的。事实上，渲染流水线中每个阶段所输出的数据往往都是作为其下个阶段的输入。例如，顶点着色器阶段（vertex shader stage）从输入装配器阶段（input assembler stage）获得输入数据，待完成相应工作后，再将结果输出至几何着色器阶段（geometry shader stage）。后续的内容将对渲染流水线的每个阶段进行更加细致的探讨。

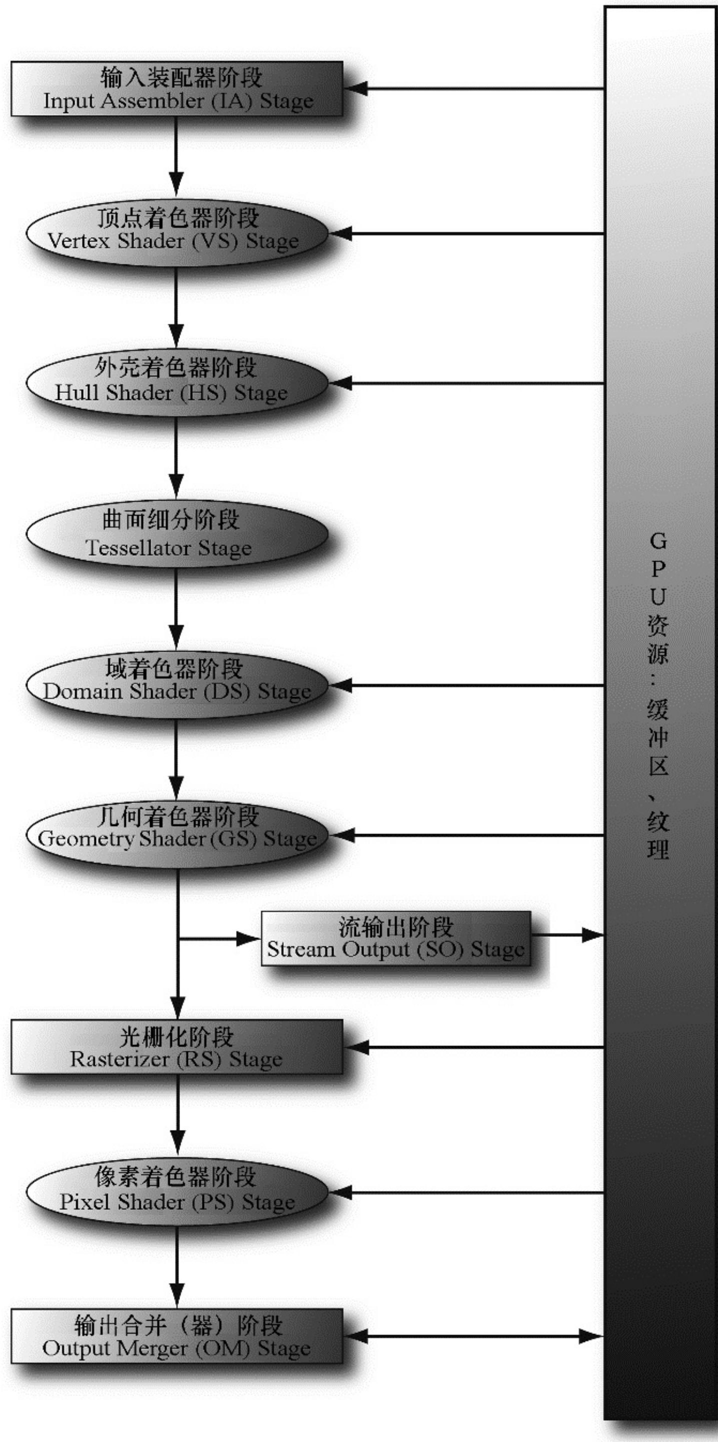


图5.11　渲染流水线的各个阶段

5.5　输入装配器阶段

输入装配器（Input Assembler，IA）阶段会从显存中读取几何数据（顶点和索引，vertex and index），再将它们装配为几何图元（geometric primitive，亦译作几何基元，如三角形和线条这种构成图形的基本元素）。这些概念将在后文中陆续介绍，但简单来说，我们是通过索引来定义如何将顶点装配在一起，从而构成图元的方法。

5.5.1　顶点

在数学中，三角形的顶点是两条边的交点；线段的顶点是它的两个端点；而对于单个的点来说，它本身就是一个顶点。

图5.12所绘的就是这几种顶点。从图示上来看，顶点似乎仅是几何图元中的一种特殊点。但是，在Direct3D中，顶点的意义却不止于此。事实上，除空间位置以外，Direct3D中的顶点还可以包含其他信息，这使我们能够利用它来表现出更为复杂的渲染效果。例如，我们将在第8章中为顶点添加法向量，以实现光照效果。而在第9章中，我们则会为顶点添加纹理坐标，从而实现纹理贴图。Direct3D为用户自定义顶点格式提供了很高的灵活性（即它允许我们定义顶点结构体中的分量），在第6章中我们会讲解与之相关的代码。届时，根据本书中需要的渲染效果，我们会依次定义几种不同的顶点格式。

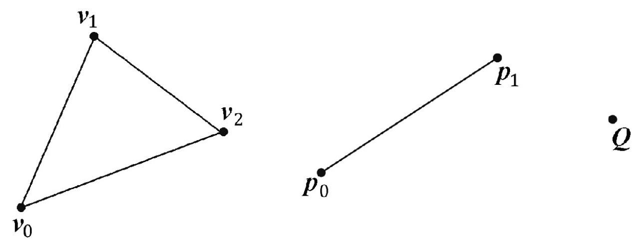


图5.12　一个由v_0、v_1、v_2三个顶点定义的三角形；
一条由顶点p_0、p_1定义的线段； 以及由顶点Q定义的点

5.5.2　图元拓扑

在Direct3D中，我们要通过一种名为顶点缓冲区（vertex buffer）的特殊数据结构，将顶点与渲染流水线相绑定。顶点缓冲区利用连续的内存来存储一系列顶点。可是，仅凭这一点并不能说明这些顶点究竟如何组成几何图元。例如，我们应将顶点缓冲区内的顶点两两一组解释成线段，还是每3个一组解释为三角形呢？对此，我们要通过指定图元拓扑（primitive topology，或称基元拓扑）来告知Direct3D如何用顶点数据来表示几何图元：

```
void ID3D12GraphicsCommandList::IASetPrimitiveTopology(
  D3D_PRIMITIVE_TOPOLOGY PrimitiveTopology);

typedef enum D3D_PRIMITIVE_TOPOLOGY
{
  D3D_PRIMITIVE_TOPOLOGY_UNDEFINED = 0,
  D3D_PRIMITIVE_TOPOLOGY_POINTLIST = 1,
  D3D_PRIMITIVE_TOPOLOGY_LINELIST = 2,
  D3D_PRIMITIVE_TOPOLOGY_LINESTRIP = 3,
  D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST = 4,
  D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP = 5,
  D3D_PRIMITIVE_TOPOLOGY_LINELIST_ADJ = 10,
  D3D_PRIMITIVE_TOPOLOGY_LINESTRIP_ADJ = 11,
  D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ = 12,
  D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP_ADJ = 13,
  D3D_PRIMITIVE_TOPOLOGY_1_CONTROL_POINT_PATCHLIST = 33,
```

```
  D3D_PRIMITIVE_TOPOLOGY_2_CONTROL_POINT_PATCHLIST = 34,
    ¦
  D3D_PRIMITIVE_TOPOLOGY_32_CONTROL_POINT_PATCHLIST = 64,
} D3D_PRIMITIVE_TOPOLOGY;
```

在用户通过命令列表（command list）修改图元拓扑之前，所有的绘制调用都会沿用当前设置的图元拓扑方式。下列代码演示的是图元拓扑的具体配置方法：

```
mCommandList->IASetPrimitiveTopology(
  D3D_PRIMITIVE_TOPOLOGY_LINELIST);
/* …通过线列表来绘制对象… */

mCommandList->IASetPrimitiveTopology(
  D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
/* …通过三角形列表来绘制对象… */

mCommandList->IASetPrimitiveTopology(
  D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);
/* …通过三角形带来绘制对象… */
```

接下来，我们将陆续解释各种不同类型的图元拓扑。除了少数情况以外，我们在本书中大多使用三角形列表。

5.5.2.1 点列表

通过枚举项D3D_PRIMITIVE_TOPOLOGY_POINTLIST来指定点列表（point list）。当使用点列表拓扑时，所有的顶点都将在绘制调用的过程中被绘制为一个单独的点，如图5.13a所示。

5.5.2.2 线条带

通过枚举项D3D_PRIMITIVE_TOPOLOGY_LINESTRIP来指定线条带

（line strip）。在使用线条带拓扑时，顶点将在绘制调用的过程中被连接为一系列的连续线段（如图5.13b所示）。所以，在这种拓扑模式下，若有$n+1$个顶点就会生成n条线段。

5.5.2.3 线列表

通过枚举项D3D_PRIMITIVE_TOPOLOGY_LINELIST来指定线列表（line list）。当使用线列表拓扑时，每对顶点在绘制调用的过程中都会组成单独的线段（如图5.13c所示）。所以$2n$个顶点就会生成n条线段。线列表与线条带的区别是：线列表中的线段可以彼此分开，而线条带中的线段则是相连的。如果线段相连的话，绘制同样数量的线段便会占用更少的顶点，因为每个处于线条带中间位置的顶点都可以同时被两条线段所共用。

5.5.2.4 三角形带

通过枚举项D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP来指定三角形带（triangle strip）。当使用三角形带拓扑时，所绘制的三角形将像图5.13d所示的那样被连接成带状。可以看到，在这种三角形连接的结构中，处于中间位置的顶点将被相邻的三角形所共同使用。因此，利用n个顶点即可生成$n-2$个三角形。

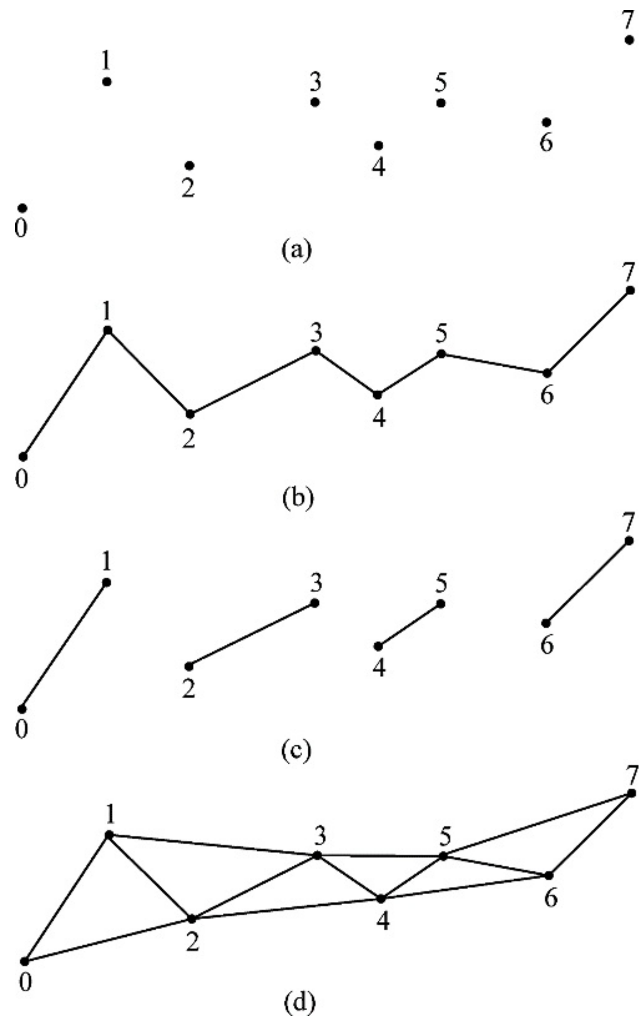


图5.13　图元拓扑（一）

（a）点列表　（b）线条带　（c）线列表　（d）三角形带

经过观察可以发现，在三角形带中，次序为偶数的三角形与次序为奇数三角形的绕序（winding order，也译作环绕顺序等，即装配图元的顶点顺序为逆时针或顺时针方向）是不同的，这就是剔除（culling，亦称消隐）问题的由来（参见5.10.2节）。为了解决这个问题，GPU内部会对偶数三角形中前两个顶点的顺序进行调换，以此使它们与奇数三角形的绕序保持一致。[8]

5.5.2.5　三角形列表

通过枚举项D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST来指定三角形列表（triangle list）。当使用三角形列表拓扑时，在绘制调用的过程中会将每3个顶点装配成独立的三角形（如图5.14a所示）；所以每$3n$个顶点会生成n个三角形。三角形列表与三角形带的区别是：三角形列表中的三角形可以彼此分离，而三角形带中的三角形则是相连的。

5.5.2.6　具有邻接数据的图元拓扑

对于存有邻接数据的三角形列表而言，每个三角形都有 3 个与之相邻的邻接三角形（adjacent triangle）。图5.14b中展示的就是这种图元拓扑。在几何着色器中，往往需要访问这些邻接三角形来实现特定的几何着色算法。为了使几何着色器可以顺利地获得这些邻接三角形的信息，我们就需要借助顶点缓冲区与索引缓冲区（index buffer）将它们随主三角形一并提交至渲染流水线。另外，此时一定要将拓扑类型指定为D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ，只有这样，渲染流水线才能得知如何以顶点缓冲区中的顶点来构建主三角形及其邻接三角形。注意，邻接图元的顶点只能用作几何着色器的输入数据，却并不会被绘制出来。即便程序没有用到几何着色器，但依旧不会绘制邻接图元。

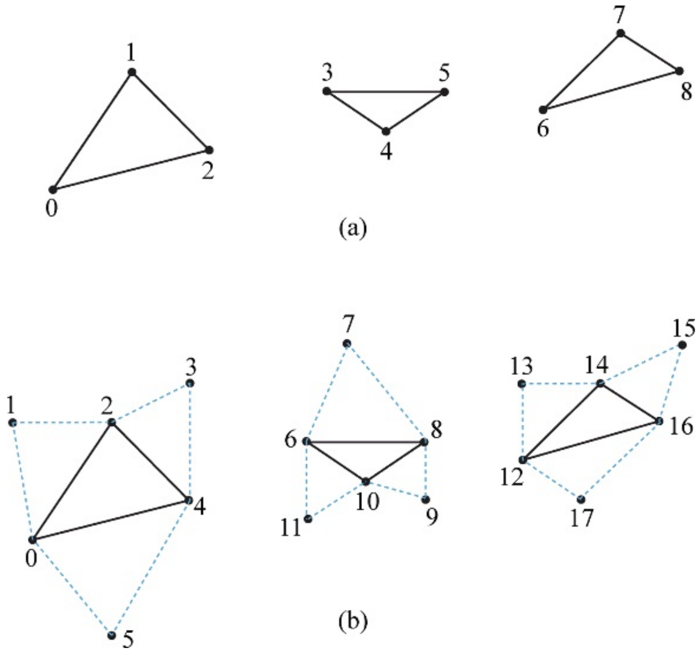


图5.14　图元拓扑（二）

（a）三角形列表　（b）具有邻接数据的三角形列表——通过观察可以发现，每个三角形共需要6个 顶点来描述它及其邻接的三角形。因此，$6n$个顶点可以生成n个三角形及其邻接数据线列表、线条带和三角形带也存在含有邻接数据的图元拓扑，相关细节可参考SDK文档

5.5.2.7　控制点面片列表

D3D_PRIMITIVE_TOPOLOGY_N_CONTROL_POINT_PATCHLIST拓扑类型表示：将顶点数据解释为具有N个控制点（control point）的面片列表（patch list）。此图元常用于渲染流水线的曲面细分阶段（tessellation stage，此环节为可选阶段），因此，我们将这种列表拓扑延至第14章中再进行讨论。

5.5.3　索引

如前所述，三角形是3D实体对象的基本组成部分。下列代码中的

两个顶点数组分别展示的是用于构建一个四边形以及一个八边形所用的三角形列表（即其中的每3个顶点将构成一个三角形）。

```
Vertex quad[6] = {
    v0, v1, v2, // 三角形0
    v0, v2, v3, // 三角形1
};

Vertex octagon[24] = {
    v0, v1, v2, // 三角形0
    v0, v2, v3, // 三角形1
    v0, v3, v4, // 三角形2
    v0, v4, v5, // 三角形3
    v0, v5, v6, // 三角形4
    v0, v6, v7, // 三角形5
    v0, v7, v8, // 三角形6
    v0, v8, v1  // 三角形7
};
```

为三角形指定顶点顺序是一项十分重要的工作，我们称这个顺序为绕序（winding order），细节可见5.10.2节。

如图5.15所示，构成3D物体的不同三角形会共用许多顶点。更具体的例子如图5.15a中构成四边形的两个三角形都使用了顶点$\boldsymbol{v}_0$和$\boldsymbol{v}_2$。因共用两个顶点而复制两个顶点数据的情况还不是太糟，但是像八边形那样的例子可就麻烦了（见图5.15b），因为每个三角形不仅都复制了一份中心顶点$\boldsymbol{v}_0$，而且此八边形边上的每个顶点都被两个三角形所同时共用。一般来说，随着模型细节和复杂度的增加，复制顶点的数量亦会急剧上升。

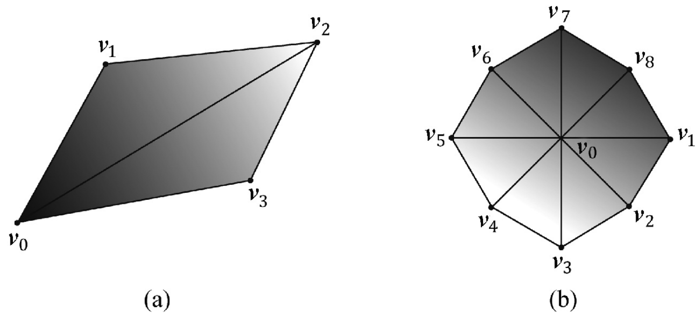


图5.15　顶点的共用

（a）由两个三角形构建的四边形　（b）通过8个三角形构建的八边形

我们不希望复制顶点数据的原因有两个：

1．增加内存的需求——为什么要多次存储同一个顶点数据呢？

2．增加图形硬件的处理负荷——为什么要多次处理同一个顶点数据呢？

借助三角形带可以在某些情况下改善顶点的复制问题，前提是这些几何体能够被组织为带状结构。但是，由于三角形列表更为灵活（该拓扑中的三角形都无需互相连接），所以值得花些心思研究一种利用三角形列表移除重复顶点的设计方案。在此，我们所采用的解决方法是使用索引（index）。整个工作流程是这样的：先创建一个顶点列表和一个索引列表。在顶点列表中收录一份所有独立的顶点，并在索引列表中存储顶点列表的索引值，这些索引定义了顶点列表中的顶点是如何组合在一起，从而构成三角形的。回顾图5.15可知，构建四边形的顶点列表如下：

```
Vertex v[4] = {v0, v1, v2, v3};
```

接下来，我们需要创建索引列表，以此来定义如何将顶点列表中的顶点组合成两个三角形。

```
UINT indexList[6] = {0, 1, 2, // 三角形0
                     0, 2, 3}; // 三角形1
```

在索引列表中，每3个元素定义了一个三角形。所以上面的索引意为：“通过顶点v[0],v[1]和v[2]来组成三角形0，再借助顶点v[0],v[2]和v[3]来构成三角形1。”

相似地，图中八边形的顶点列表构造如下：

```
Vertex v[9] = {v0, v1, v2, v3, v4, v5, v6, v7, v8};
```

相应的索引列表是：

```
UINT indexList[24] = {
  0, 1, 2, // 三角形0
  0, 2, 3, // 三角形1
  0, 3, 4, // 三角形2
  0, 4, 5, // 三角形3
  0, 5, 6, // 三角形4
  0, 6, 7, // 三角形5
  0, 7, 8, // 三角形6
  0, 8, 1  // 三角形7
};
```

待处理完顶点列表中那些独立的顶点之后，显卡就能通过索引列表把顶点组合成一系列三角形。可以看到，我们已经将“复用的顶点数据”转化为索引列表，但是这样做的效果要比之前的方法更好，这是因为：

1．索引皆是简单的整数，不会像使用整个顶点结构体那样占用更

多的内存（而且，随着顶点结构体中分量的不断增多，将会使内存的需求变得更为急迫）。

2．若辅以适当的顶点缓存排序，则图形硬件将不必再次处理重复使用的顶点，从缓存中直接取得即可（这种情况十分普遍）[9]。

5.6　顶点着色器阶段

待图元被装配完毕后，其顶点就会被送入顶点着色器阶段（vertex shader stage，简记作VS）。我们可以把顶点着色器看作一种输入与输出数据皆为单个顶点的函数。每个要被绘制的顶点都须经过顶点着色器的处理再送往后续阶段。事实上，我们可以认为在硬件中执行的是下列处理过程：

```
for(UINT i = 0; i < numVertices; ++i)
  outputVertex[i] = VertexShader( inputVertex[i] );
```

其中的顶点着色器函数（VertexShader）就是我们要实现的那一部分，因为在这一阶段中对顶点的操作实际是由GPU来执行的，所以速度很快。

我们可以利用顶点着色器来实现许多特效，例如变换、光照和位移贴图（displacement mapping，也译作置换贴图。map有映射之意，因此也有译作位移映射，类似的还有在后面将见到的纹理贴图、法线贴图等）。请牢记：在顶点着色器中，不但可以访问输入的顶点数据，也能够访问纹理和其他存于显存中的数据（如变换矩阵与场景的光照信息）。

我们将在全书中看到各种不同的顶点着色器示例，所以在完成本书的学习后，我们应当会对它可以实现的具体功能有一个深刻的认识。在本书的第一个代码示例中，我们仅用顶点着色器对顶点进行变换处理。

接下来，我们将介绍几种常用的空间变换。

5.6.1　局部空间和世界空间

试想我们正在为一部电影而奔忙，而我们所在的团队必须为一些特效镜头打造一个与火车有关的微缩场景。其中，我们的具体任务是制作一架袖珍小桥。当然，我们并不会把桥直接搭建在场景之中，否则，便需要在一个极其复杂的环境中小心翼翼地操作，以防破坏场景中的其他景物，一旦失手便会功亏一篑。相对来讲，我们更加愿意在远离场景的工作室内制作此模型。待大功告成之后，才会把它以恰当的角度置于场景中合适的位置。

3D美工们在构筑3D对象时也在做着类似的事情。他们并不会在全局场景坐标系（即世界空间，也译为世界坐标系，world space）中构建物体的几何形状，而是相对于局部坐标系（局部空间，也译为局部坐标系，local space）来创建物体。局部坐标系通常是一种以目标物体的中心为原点（也有例外，视具体情况而定），并且坐标轴与该物体对齐的简易便用坐标系[10]。只要在局部空间中定义了3D模型的各顶点，我们就能将它变换至全局场景之中。为了做到这一点，我们还必须定义局部空间与世界空间两者的联系。具体的实现方法是：根据物体的位置与朝向，指定其局部空间坐标系的原点和诸坐标轴相对于全局场景坐标系的坐标，再运用坐标变换即可将物体从局部空间变换至世界空间（参见图5.16并重温3.4节）。将局部坐标系内的坐标转换到全局场景坐标系中的过程叫作世界变换（world transform），所使用的变换矩阵名为世界矩

阵（world matrix）。由于场景中每个物体的朝向和位置都可能各不相同，因此它们都有自己特定的世界矩阵。当每个物体都从各自的局部空间变换到世界空间后，它们的坐标都将位于同一坐标系（即世界坐标系）之中。如果希望直接在世界空间内定义一个物体，那么就可以使用单位世界矩阵（identity world matrix）。

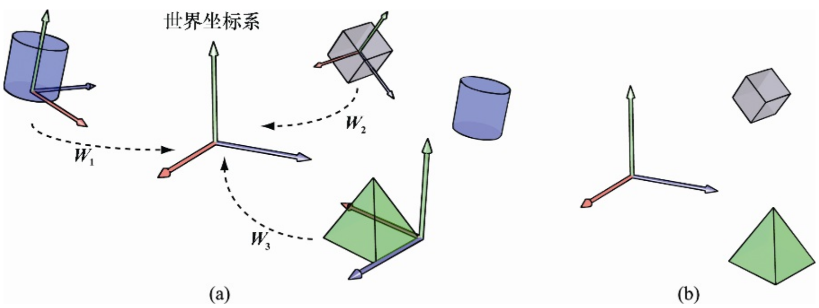


图5.16　图a中，每一个物体都相对于自己的局部坐标系来定义各自的顶点。另外，我们还要根据 物体在场景中的具体方位来相对于世界空间坐标系定义每个局部坐标系的位置与朝向。接下来， 再通过坐标变换把所有物体的坐标都变换至世界空间坐标系。图b中，完成世界变换之后， 物体的顶点坐标都将统一用世界坐标系来表示

在每个3D模型各自的局部坐标系中来定义它们有若干优点：

1．易于使用。例如，物体的中心通常位于局部空间中的原点，并且关于主轴对称。举个例子，当我们采用一个以立方体中心作为原点，且坐标轴正交于各立方体面的局部坐标系时，我们就能轻而易举地确定出立方体的各个顶点。整个过程如图5.17所示。

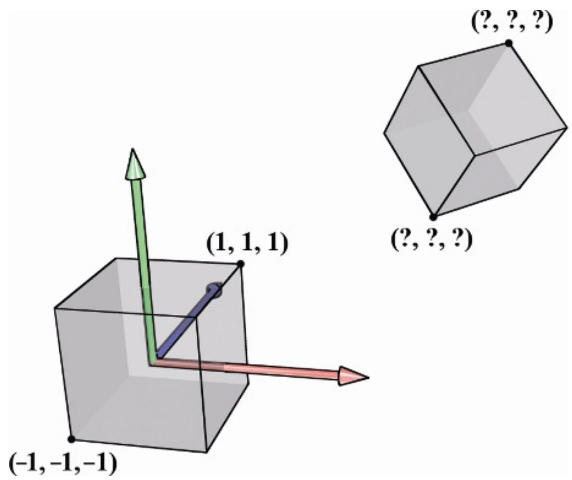


图5.17　当立方体与坐标系的主轴对齐且其中心位于坐标系的原点之时，它的顶点是十分易于确定的。反之，当一个立方体处于坐标系中任意的位置以及朝向的时候，要确定它的顶点坐标就比较困难了。因此，在构造物体的时候，我们一般总是选择以物体的中心为原点且轴对齐于此物体的简便坐标系

2．物体应当可以跨越多个场景而重复使用，所以将物体坐标相对于某个特定场景进行硬编码并不是明智之举。更好的办法则是保存物体相对于局部坐标系的顶点坐标，再通过坐标变换矩阵去定义每个场景中局部坐标系与世界坐标系的关系。

3．最后一点，我们有时可能需要在场景中多次绘制同一个物体，但是它们的位置、方向和大小却各不相同（例如，将一个树形对象绘制多次来构成一片森林）。如若在每次创建物体实例时都要复制它的顶点和索引数据，将极其消耗资源。因此，我们通常的做法是存储一份几何体相对于其局部空间的副本（即该几何体的顶点列表和索引列表）。接着，按所需次数来绘制此物体，每次辅以不同的世界矩阵来指定物体在世界空间中的位置、方向和大小。这种方法称为实例化（instancing）。

正如3.4.3节中所讲，世界矩阵描述着物体局部空间与世界空间的联

系，而列于矩阵内的行向量则是下面要提到的坐标。如果 $\boldsymbol{Q}_w = (Q_x, Q_y, Q_z, 1)$，$\boldsymbol{u}_w = (u_x, u_y, u_z, 0)$，$\boldsymbol{v}_w = (v_x, v_y, v_z, 0)$和 $\boldsymbol{w}_w = (w_x, w_y, w_z, 0)$分别描述了局部空间内的原点、$x$轴、$y$轴和$z$轴相对于世界空间的齐次坐标，那么由3.4.3节中的内容可知，从局部空间至世界空间的坐标变换矩阵为：

$$\boldsymbol{W} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_{\mathrm{y}} & w_z & 0 \\ Q_x & Q_y & Q_z & 1 \end{bmatrix}$$

可以看到，为了构建一个世界矩阵，我们必须弄清局部空间中原点和各坐标轴相对于世界空间的坐标关系。但这样做有时并不简单亦不直观。一种更常用的办法是定义一系列的变换组合$\boldsymbol{W}$，即$\boldsymbol{W} = \boldsymbol{SRT}$。首先，缩放矩阵$\boldsymbol{S}$将物体缩放到世界空间；其次，旋转矩阵$\boldsymbol{R}$用来定义局部空间相对于世界空间的朝向，最后，平移矩阵$\boldsymbol{T}$定义的是局部空间的原点相对于世界空间的位置。重温3.5节可知，我们能够将这一系列变换视为一种坐标变换，而矩阵$\boldsymbol{W} = \boldsymbol{SRT}$中的行向量则分别存储的是局部空间的$x$轴、$y$轴、$z$轴及原点相对于全局空间的的齐次坐标。

示例

假设我们在局部空间定义了一个单位正方形，其最小点和最大点的坐标分别为(−0.5, 0, −0.5)与(0.5, 0, 0.5)。现在要求出一个世界矩阵，使此正方形在世界空间中的边长为2，在世界空间xz平面内顺时针旋转$45°$，且中心位于世界空间的坐标(10, 0, 10)处。据此，我们构造矩阵$\boldsymbol{S}$、$\boldsymbol{R}$

、$\boldsymbol{T}$及$\boldsymbol{W}$的过程如下：

$$\boldsymbol{S}=\begin{bmatrix}2&0&0&0\\0&1&0&0\\0&0&2&0\\0&0&0&1\end{bmatrix}\boldsymbol{R}=\begin{bmatrix}\sqrt{2}/2&0&-\sqrt{2}/2&0\\0&1&0&0\\\sqrt{2}/2&0&\sqrt{2}/2&0\\0&0&0&1\end{bmatrix}\boldsymbol{T}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\10&0&10&1\end{bmatrix}$$

$$\boldsymbol{W}=\boldsymbol{SRT}=\begin{bmatrix}\sqrt{2}&0&-\sqrt{2}&0\\0&1&0&0\\\sqrt{2}&0&\sqrt{2}&0\\10&0&10&1\end{bmatrix}$$

从第3.5节中可知，矩阵$\boldsymbol{W}$中的行向量描述了此正方形局部坐标系的诸坐标轴与原点相对于世界空间的坐标，即有$\boldsymbol{u}_W=(\sqrt{2},0,-\sqrt{2},0)$，$\boldsymbol{v}_W=(0,1,0,0)$，$\boldsymbol{w}_W=(\sqrt{2},0,\sqrt{2},0)$和$\boldsymbol{Q}_W=(10,0,10,1)$。如果我们利用矩阵$\boldsymbol{W}$将此局部空间向世界空间进行坐标变换，则最终会把正方形置于题设中所期望的世界空间内的预定位置（见图5.18）。

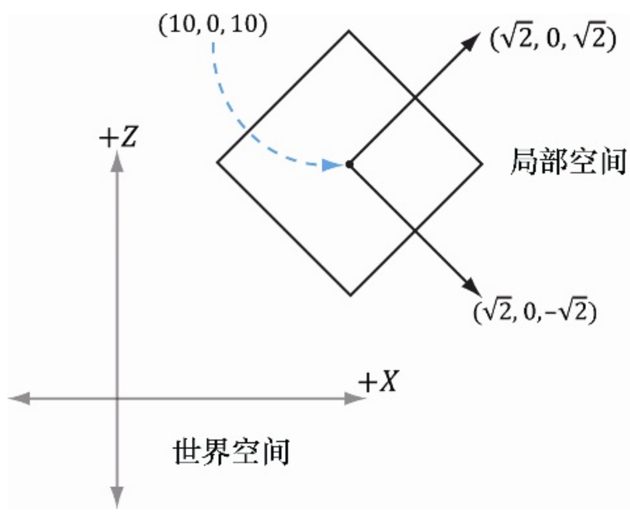


图5.18　世界矩阵中行向量所描述的是局部坐标系的原点与众坐标轴相对于世界坐标系的坐标

$$[-0.5,\ 0,\ -0.5,\ 1]\boldsymbol{W}=[10-\sqrt{2},\ 0,\ 0,\ 1]$$

$$[-0.5,\ 0,\ +0.5,\ 1]\boldsymbol{W}=[0,\ 0,\ 10+\sqrt{2},\ 1]$$

$$[+0.5,\ 0,\ +0.5,\ 1]\boldsymbol{W} = [10+\sqrt{2},\ 0,\ 0,\ 1]$$

$$[+0.5,\ 0,\ -0.5,\ 1]\boldsymbol{W} = [0,\ 0,\ 10-\sqrt{2},\ 1]$$

此例的亮点是在不指明$\boldsymbol{Q}_W$、$\boldsymbol{u}_W$、$\boldsymbol{v}_W$和$\boldsymbol{w}_W$的情况下，直接通过复合一系列简单的变换来建立世界矩阵。这通常比用$\boldsymbol{Q}_W$、$\boldsymbol{u}_W$、$\boldsymbol{v}_W$和$\boldsymbol{w}_W$求取世界矩阵的方法要容易得多，因为我们只需了解物体在世界空间中的大小、朝向以及位置即可。

另一种考虑世界变换的观点是把局部空间坐标当作世界空间坐标来看待（此方法就相当于用单位矩阵进行世界变换）。这样一来，如果在物体局部空间的原点处建模，那么该物体也就位于世界空间的原点处。通常来说，我们不大可能把物体全都建立在世界空间的原点处。所以往往还是要为每个物体运用一系列变换，使之缩放、旋转，并令其位于世界空间中的预定位置。从数学角度上来讲，这种变换与由局部空间转换至世界空间所用的坐标变换矩阵进行的是同一种世界变换。

5.6.2 观察空间

为了构建场景的2D图像，我们必须在场景中架设一台虚拟摄像机。该摄像机确定了观察者可见的视野，也就是生成2D图像所需的场景空间范围。对此，我们先为该摄像机赋予一个图5.19所示的局部坐标系（这被称作观察空间（view space），也译作观察坐标系、视图空间、视觉空间（eye space）或摄像机空间（camera space））。在此坐标系中，该虚拟摄像机位于原点并沿z轴的正方向观察，x轴指向摄像机

的右侧，y轴则指向摄像机的上方。与相对于世界空间来描述场景中的物体顶点不同，观察空间用于在渲染流水线的后续阶段中描述这些顶点相对于摄像机坐标系的坐标。由世界空间至观察空间的坐标变换称为取景变换（view transform，也译作观察变换、视图变换等），此变换所用的矩阵则称为观察矩阵（view matrix，亦译作视图矩阵）。

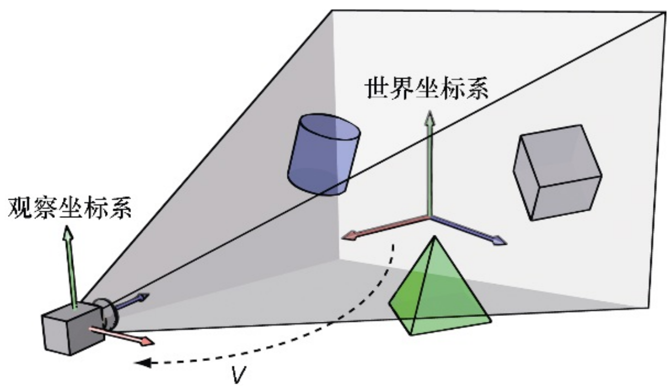


图5.19　将世界空间中的顶点坐标变换至摄像机空间

如果$\boldsymbol{Q}_W=(\boldsymbol{Q}_x,\boldsymbol{Q}_y,\boldsymbol{Q}_z,1)$，$\boldsymbol{u}_W=(u_x,u_y,u_z,0)$，$\boldsymbol{v}_W=(v_x,v_y,v_z,0)$以及$\boldsymbol{w}_W=(w_x,w_y,w_z,0)$，分别表示了观察空间中原点、$x$轴、$y$轴和$z$轴相对于世界空间的齐次坐标。那么，根据3.4.3节中的内容可知，由观察空间到世界空间的坐标变换矩阵为：

$$\boldsymbol{W}=\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & Q_z & 1 \end{bmatrix}$$

然而这并不是我们所期待的变换。刚好相反，我们需要的是从世界空间到观察空间的这一逆变换。回顾3.4.5节可知，逆变换可由变换矩阵的逆来求得，所以从世界空间到观察空间的坐标变换矩阵为$\boldsymbol{W}^{-1}$。

世界坐标系和观察坐标系通常只有位置和朝向这两点差异，所以由观察空间到世界空间的变换可以直接表示为$\boldsymbol{W}=\boldsymbol{RT}$（即世界矩阵可以分解为一个旋转矩阵与一个平移矩阵的乘积）。此形式使得上述逆变换更易于计算：

$$\boldsymbol{V}=\boldsymbol{W}^{-1}=(\boldsymbol{RT})^{-1}=\boldsymbol{T}^{-1}\boldsymbol{R}^{-1}=\boldsymbol{T}^{-1}\boldsymbol{R}^{T}=$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -Q_x & -Q_y & -Q_z & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ -\boldsymbol{Q}\cdot\boldsymbol{u} & -\boldsymbol{Q}\cdot\boldsymbol{v} & -\boldsymbol{Q}\cdot\boldsymbol{w} & 1 \end{bmatrix}$$

因此，观察矩阵形为：

$$\boldsymbol{V}=\begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ -\boldsymbol{Q}\cdot\boldsymbol{u} & -\boldsymbol{Q}\cdot\boldsymbol{v} & -\boldsymbol{Q}\cdot\boldsymbol{w} & 1 \end{bmatrix}$$

现在我们来展示一种用以构建观察矩阵中诸向量的直观方法。设$\boldsymbol{Q}$为虚拟摄像机的位置，$\boldsymbol{T}$为此摄像机对准的观察目标点（target point）。接下来，设$\boldsymbol{j}$为表示世界空间“向上”方向的单位向量。（在本书中，我们用世界空间中的平面xo作为场景中的“地平面”，并以世界空间的y轴来指示场景内“向上”的方向。因此，$\boldsymbol{j}=(0,1,0)$仅是平行于世界空间中y轴的一个单位向量。有时为了方便起见，一些应用程序也可能选择平面xy作为地平面，而选z轴来指示“向上”的方向）。对于图5.20来讲，虚拟摄像机的观察方向为：

$$\boldsymbol{w}=\frac{\boldsymbol{T}-\boldsymbol{Q}}{\|\boldsymbol{T}-\boldsymbol{Q}\|}$$

该向量表示虚拟摄像机局部空间的z轴。指向$\boldsymbol{w}$“右侧”的单位向量为：

$$\boldsymbol{u} = \frac{\boldsymbol{j} \times \boldsymbol{w}}{\|\boldsymbol{j} \times \boldsymbol{w}\|}$$

它表示的是虚拟摄像机局部空间的x轴。最后，该摄像机局部空间的y轴为：

$$\boldsymbol{v} = \boldsymbol{w} \times \boldsymbol{u}$$

因为$\boldsymbol{w}$和$\boldsymbol{u}$为互相正交的单位向量，所以$\boldsymbol{w} \times \boldsymbol{u}$亦必为单位向量。由此，我们也就无须对向量$\boldsymbol{v}$进行规范化处理了。

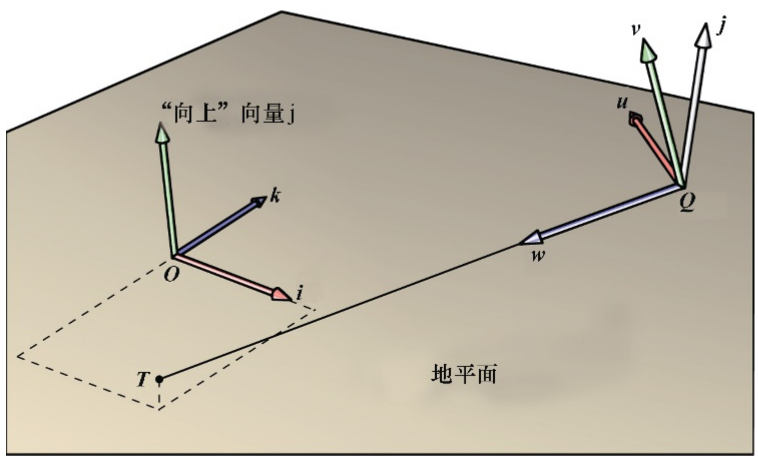


图5.20　根据指定的摄像机位置、观察的目标点以及世界空间的“向上”向量来构建摄像机坐标系

综上所述，只要给定摄像机的位置、观察目标点以及世界空间中“向上”方向的向量，我们就能构建出对应的摄像机局部坐标系，并推导出相应的观察矩阵。

DirectXMath库针对上述计算观察矩阵的处理流程提供了以下函

数：

```
XMMATRIX XM_CALLCONV XMMatrixLookAtLH(      // 输出观察矩阵V
  FXMVECTOR EyePosition,                    // 输入虚拟摄像机位置Q
  FXMVECTOR FocusPosition,                  // 输入观察目标点T
  FXMVECTOR UpDirection);                   // 输入世界空间中向上方向的向量j
```

一般来说，世界空间中的y轴方向与虚拟摄像机“向上”向量的方向相同，所以，我们通常将“向上”向量定为$\boldsymbol{j} = (0,1,0)$。举个例子，假设我们希望把虚拟摄像机架设于世界空间内点(5, 3, −10)的位置，并令它观察世界空间的原点（0, 0, 0），则构建相应观察矩阵的过程为：

```
XMVECTOR pos   = XMVectorSet(5, 3, -10, 1.0f);
XMVECTOR target = XMVectorZero();
XMVECTOR up    = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);

XMMATRIX V = XMMatrixLookAtLH(pos, target, up);
```

5.6.3 投影和齐次裁剪空间

前一节我们讲解了摄像机在世界空间中的位置和朝向，除此之外，它还有另一个关键组成要素：即摄像机可观察到的空间体积（volume of space）。此范围可用一个由四棱锥截取的平截头体（frustum，即四棱台）来表示（如图5.21所示）。

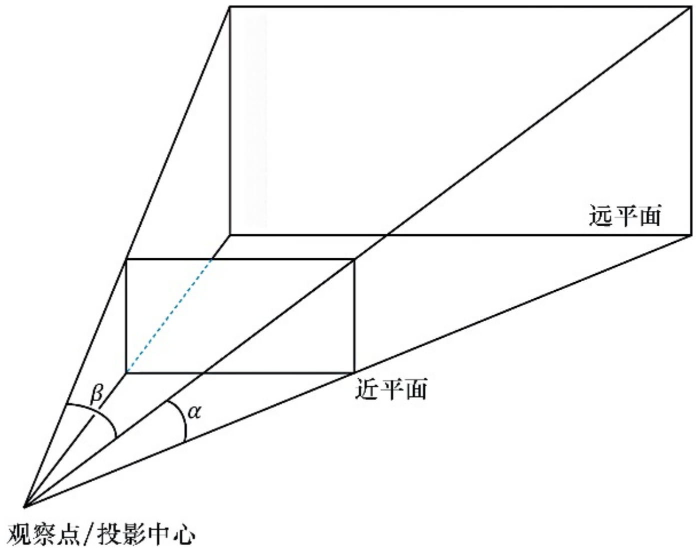


图5.21　定义摄像机所“观察”空间体积的平截头体

下一个任务是：将平截头体内的3D几何体投影到一个2D投影窗口（projection window）之中。根据前文所述的透视投影（perspective projection）的原理可知，投影必将沿众平行线汇聚于消失点上，而且随着物体3D深度的增加，其投影的尺寸也将逐渐变小（见图5.22）。我们将由顶点到观察点（eye point，也译作视点）的连线称为顶点的投影线（vertex’s line of projection）。继而就可以定义出：将3D顶点$\mathbf{v}$变换至其投影线与2D投影平面交点$\mathbf{v}'$的透视投影变换（perspective projection transformation）。我们称点$\mathbf{v}'$为点$\mathbf{v}$的投影。3D物体的投影即为构成该物体上所有顶点的投影。

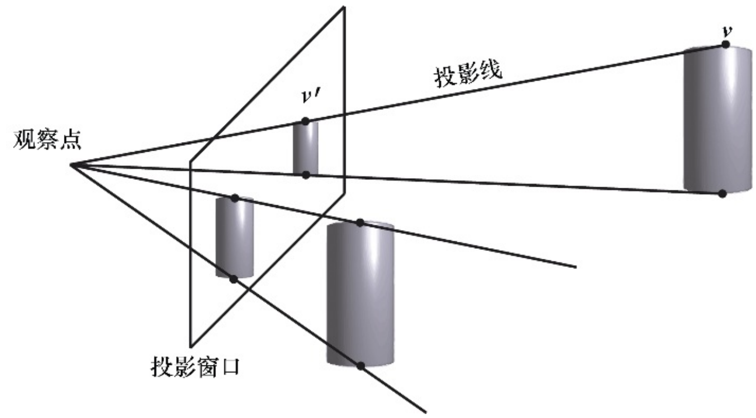


图5.22　在3D空间中，两个大小相同，但却处于不同3D深度的圆柱体。靠近观察点的圆柱体投影 要大于远离观察点的圆柱体投影。位于平截头体中的几何体会被投射到投影窗口内，而平截头体 之外的几何体则会被投射到投影平面上，投影窗口以外的区域

5.6.3.1　定义平截头体

在观察空间中，我们可以通过近平面（near plane，也译作近裁剪面）n、远平面（far plane，也译作远裁剪面）f、垂直视场角（vertical field of view angle）α以及纵横比（aspect ratio，也作宽高比）r这4个参数来定义一个：以原点作为投影的中心，并沿z轴正方向进行观察的平截头体（可参见图5.23）。值得注意的是，位于观察空间中的远、近平面皆平行于平面xy，因此，我们就能方便地确定出它们分别沿z轴到原点的距离。纵横比的定义为$r = w/h$，其中w为投影窗口的宽度，h为投影窗口的高度（以观察空间的单位为准）。投影窗口实质上即为观察空间中场景的2D图像。由于该图像终将被映射到后台缓冲区中，因此，我们希望令投影窗口与后台缓冲区两者的纵横比保持一致。为此，我们通常将投影窗口的纵横比指定为后台缓冲区的纵横比（比值并没有单位）。例如，假设后台缓冲区的大小为800×600，那么我们就指定

$r = \frac{800}{600} \approx 1.333$。如若投影窗口与后台缓冲区的纵横比不一致，那么映射的过程中，就需要对投影窗口在将投影窗口进行不等比缩放（non-uniform scaling，也有将其译为非均匀缩放，非一致性缩放或非统一缩放），继而导致图像出现拉伸变形的现象（例如，投影窗口中的圆形映射到后台缓冲区时，可能会被拉伸为椭圆形）。

我们现在通过垂直视场角α和纵横比r来确定水平视场角（horizontal field of view angle）β。为此，给出相关示意图5.23。注意，投影窗口的实际大小并不重要，关键在于确定纵横比。因此，出于方便，我们将高定为2，而宽则必满足：

$$r = \frac{w}{h} = \frac{w}{2} \Rightarrow w = 2r$$

为了求出具体的垂直视场角α，我们假定投影窗口到原点的距离为d：

$$\tan\left(\frac{\alpha}{2}\right) = \frac{1}{d} \Rightarrow d = \cot\left(\frac{\alpha}{2}\right)$$

因此，当投影窗口的高度为2且垂直视场角为α时，我们就能确定该投影窗口沿z轴到观察点的距离d。已知这些条件，即可求取水平视场角β。观察图5.23中的平面xz可以发现：

$$\begin{aligned}\tan\left(\frac{\beta}{2}\right) &= \frac{r}{d} = \frac{r}{\cot\left(\frac{\alpha}{2}\right)} \\ &= r \cdot \tan\left(\frac{\alpha}{2}\right)\end{aligned}$$

所以，一旦给定垂直视场角α和纵横比r，我们必能求出水平视场角

β：

$$\beta = 2\arctan\left(r \cdot tan\left(\frac{\alpha}{2}\right)\right)$$

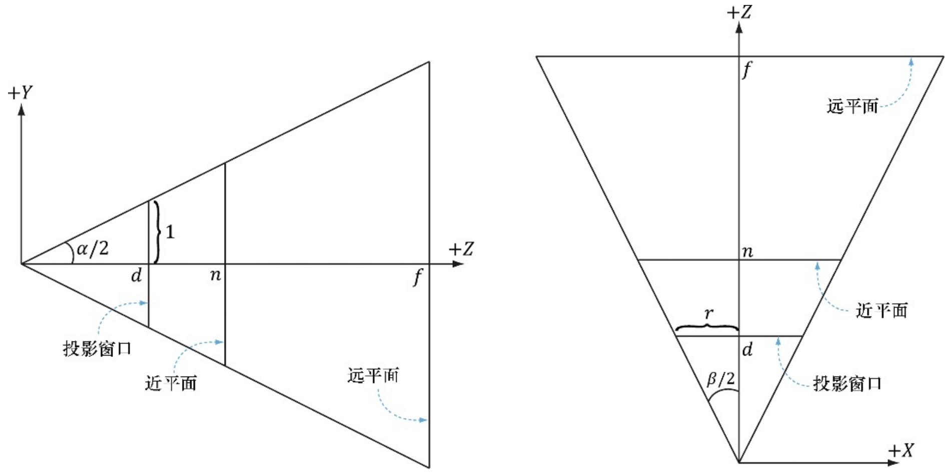


图5.23　根据指定的垂直视场角α和纵横比r来推导水平视场角β

5.6.3.2　投影顶点

我们希望求出给定点（x, y, z）在投影平面$z = d$中的投影（x', y', d），见图5.24。通过在x轴和y轴上分别利用相似三角形的性质，我们可以发现：

$$\frac{x'}{d} = \frac{x}{z} \Rightarrow x' = \frac{xd}{z} = \frac{xcot(\alpha/2)}{z} = \frac{x}{ztan(\alpha/2)}$$

与

$$\frac{y'}{d} = \frac{y}{z} \Rightarrow y' = \frac{yd}{z} = \frac{ycot(\alpha/2)}{z} = \frac{y}{ztan(\alpha/2)}$$

同时不难看出，若点(x, y, z)位于平截头体内，当且仅当：

$$-r \leqslant x' \leqslant r$$

$$-1 \leqslant y' \leqslant 1$$

$$n \leqslant z \leqslant f$$

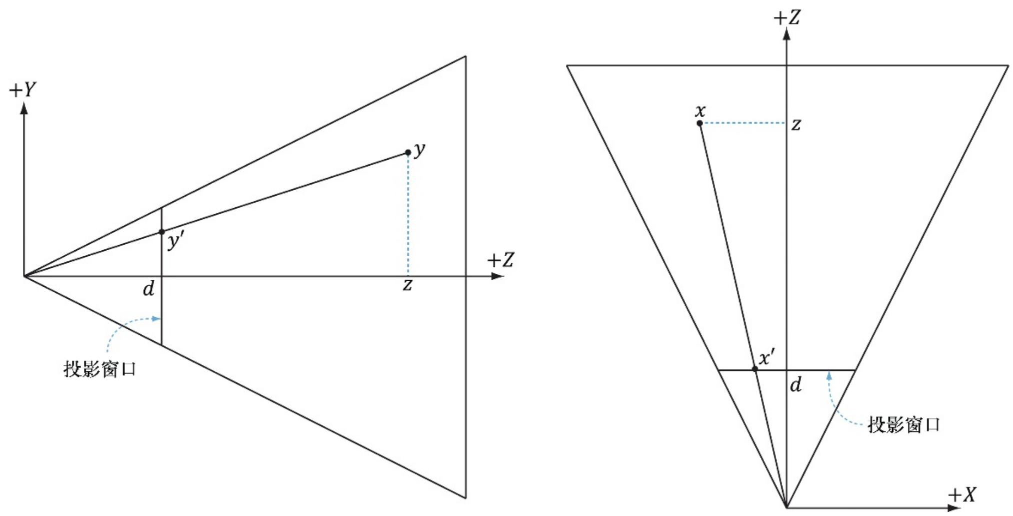


图5.24　利用相似三角形的性质来求取目标点在投影平面内的投影

5.6.3.3　规格化设备坐标

在前一小节里，我们讨论了如何在观察空间中计算诸点的投影坐标。在计算过程中，要保证观察空间里投影窗口的高为2，宽为$2r$，这r即为纵横比。但是，如果投影窗口的尺寸依赖于纵横比便会产生一个问题：由于硬件会涉及一些与投影窗口大小有关的操作（诸如将投影窗口映射到后台缓冲区等），这意味着我们还需要将纵横比告知硬件。因此，如果能去除投影窗口对纵横比的依赖，那么处理过程会更加简单。对此，我们的解决办法是将x坐标上的投影区间从$[-r, r]$缩放至归一化区间$[-1, 1]$，就像下面这样：

$$-r \leqslant x' \leqslant r$$

$$-1 \leqslant \frac{x'}{r} \leqslant 1$$

经此映射处理后，x坐标和y坐标就成为了规格化设备坐标（Normalized Device Coordinates，NDC）（请注意，这里并没有对z坐标进行归一化处理）。此时，若点(x, y, z)位于平截头体之中，当且仅当：

$$-1 \leqslant x'/r \leqslant 1$$

$$-1 \leqslant y' \leqslant 1$$

$$n \leqslant z \leqslant f$$

我们可以把由观察空间到NDC空间的变换视为一种单位换算（unit conversion）。观察两者的转换过程，可知存在如下关系：在x轴上，1个NDC单位等于观察空间中的r个单位（即1 ndc = r vs）。所以，若给出x个观察空间单位，我们就可以根据上述关系将它转换为NDC单位（式中的vs即观察空间view space的缩写）：

$$x \text{ vs} \cdot \frac{1 \text{ ndc}}{r \text{ vs}} = \frac{x}{r} ndc$$

我们可以基于上式修改投影公式，从而直接求出以NDC坐标来表示的x轴和y轴上的投影坐标：

$$x' = \frac{x}{rz\tan(\alpha/2)}$$

$$y' = \frac{y}{z\tan(\alpha/2)} \qquad (5.1)$$

注意，在NDC坐标中，投影窗口的高和宽都为2，所以它的大小是固定的，硬件也就无须知道纵横比。但是，我们一定要确保将投影坐标映射到NDC空间内（图形硬件假设我们会完成这项工作）。

5.6.3.4 用矩阵来表示投影公式

为了保证变换的一致性，我们将用矩阵来表示投影变换。然而，由于式（5.1）的非线性特征，所以并不存在与之对应的矩阵表示。我们的“诀窍”是将其“一分为二”：即将其分为线性与非线性两个处理部分。非线性部分要进行除以z的计算过程。正如在下节中所讨论的，我们还将对z坐标进行归一化处理；这就意味着在执行非线性部分除以z的计算时，我们却无最初的z坐标可用。也就是说，我们一定要在此变换之前保存早先传入的初始z坐标。为了做到这一点，我们要利用齐次坐标将输入的z坐标复制到输出的w坐标。根据矩阵的乘法运算法则，需要令元素$[2][3]=1$以及元素$[3][3]=0$来加以实现（这里采用的是以0为基准的索引）。此投影矩阵形如：

$$\boldsymbol{P}=\begin{bmatrix}\frac{1}{r\tan(\alpha/2)} & 0 & 0 & 0\\ 0 & \frac{1}{\tan(\alpha/2)} & 0 & 0\\ 0 & 0 & A & 1\\ 0 & 0 & B & 0\end{bmatrix}$$

可以看到，我们在矩阵当中设置了常量A和常量B（在下一节中会推导它们的定义），利用它们即可把输入的z坐标变换到归一化范围。令任意点$(x,y,z,1)$与该矩阵相乘将会得到：

$$[x,\ y,\ z,\ 1]\begin{bmatrix}\frac{1}{r\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 & 0\\ 0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0\\ 0 & 0 & A & 1\\ 0 & 0 & B & 0\end{bmatrix} = \left[\frac{x}{r\tan\left(\frac{\alpha}{2}\right)}, \frac{y}{\tan\left(\frac{\alpha}{2}\right)}, Az+B, z\right] \quad (5.2)$$

在顶点与投影矩阵相乘之后（即线性部分），我们还要通过将每个坐标分别除以$w = z$（即非线性部分）来完成整个变换过程：

$$\left[\frac{x}{r\tan\left(\frac{\alpha}{2}\right)}, \frac{y}{\tan\left(\frac{\alpha}{2}\right)}, Az+B, z\right] \xrightarrow{\text{除以}w} \left[\frac{x}{rz\tan\left(\frac{\alpha}{2}\right)}, \frac{y}{z\tan\left(\frac{\alpha}{2}\right)}, A+\frac{B}{z}, 1\right] \qquad (5.3)$$

顺便讲一下，部分读者可能会怀疑是否会发生除以0的情况。然而，事实上即使是近平面的z值也应当大于0，所以处于这一位置上的点将会被裁剪（5.9节）掉。除以w的计算过程有时被称为透视除法（perspective divide）或齐次除法（homogeneous divide）。可以看出，此式中的x、y投影坐标也与式（5.1）中的一致。

5.6.3.5　归一化深度值

待投影操作完毕后，所有的投影点都会位于2D投影窗口上，从而构成视觉上可见的2D图像。看起来，我们似乎在此时就可以丢弃原始的3D z坐标了。然而，为了实现深度缓冲算法，我们仍需保留这些3D深度信息。就像Direct3D希望将x、y坐标映射到归一化范围一样，深度坐标也要被映射到归一化区间[0, 1]以内。因此，我们必须构建一个保序（order preserving）函数$g(z)$，用来把z坐标从区间$[n, f]$映射到区间$[0, 1]$。由于该函数具有保序性，即如果$z_1, z_2 \in [n, f]$且$z_1 < z_2$，那么$g(z_1) < g(z_2)$。也就是说，对深度值进行归一化处理后，深度关系保持不变。所以，在实现深度缓冲算法的过程中，我们仍能在归一化区间内正确地比较出不同点之间的深度关系。

虽然通过一次缩放和平移操作，便能将z坐标从$[n, f]$区间映射到[0, 1]区间。但是此方法却不能整合到我们当前的投影方案中去。从式（5.3）中可以看出，z坐标将经过以下变换的处理：

$$g(z) = A + \frac{B}{z}$$

现在，我们需要根据下列约束求出对应的A与B：

条件1：$g(n) = A + B/n = 0$（将近平面映射为0）；

条件2：$g(f) = A + B/f = 1$（把远平面映射为1）。

根据条件1解得B：

$$B = -An$$

将B代入条件2，解出A

$$A + \frac{-An}{f} = 1$$

$$\frac{Af - An}{f} = 1$$

$$Af - An = f$$

$$A = \frac{f}{f - n}$$

所以，

$$g(z) = \frac{f}{f - n} - \frac{nf}{(f - n)z}$$

根据函数g的图像（见图5.25）可以看出，它是严格递增（保序性）的非线性函数。同时，这也反映了$g(z)$大部分取值是由近平面附近的深度值所计算得出的。换言之，大多数的深度值被集中地映射到了取值区间中的一段较小的区域内。这将引发深度缓冲区的精度问题（由于计算机表示的数值范围有限，使计算机不足以区分归一化深度值之间的微小差异）。对此，我们一般建议令近平面与远平面尽可能地接近，以改善深度值的精度问题。

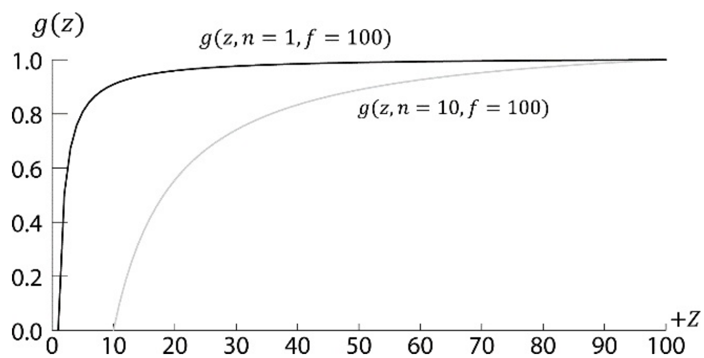


图5.25　不同近平面的$g(z)$函数图像

既然已经求得了A和B，我们就可以确定出完整的透视投影矩阵（perspective projection matrix）：

$$\boldsymbol{P}=\begin{bmatrix} \frac{1}{r\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & \frac{-nf}{f-n} & 0 \end{bmatrix}$$

在顶点乘以投影矩阵之后但还未进行透视除法之前，几何体会处于所谓的齐次裁剪空间（homogeneous clip space）或投影空间（projection space）之中。待完成透视除法之后，便是用规格化设备坐标（NDC）

来表示几何体了。

5.6.3.6 XMMatrixPerspectiveFovLH函数

我们可以利用DirectXMath库内的XMMatrixPerspectiveFovLH函数来构建透视投影矩阵：

```
// 返回投影矩阵
XMMATRIX XM_CALLCONV XMMatrixPerspectiveFovLH(
  float FovAngleY,    // 用弧度制表示的垂直视场角
  float Aspect,       // 纵横比 = 宽度 / 高度
  float NearZ,        // 到近平面的距离
  float FarZ);        // 到远平面的距离
```

下面的代码片段详细解释了XMMatrixPerspectiveFovLH函数的用法。在此例中，我们将垂直视场角指定为$45°$，近平面位于$z = 1$处，远平面位于$z = 1000$处（这些长度皆以观察空间中的单位表示）。

```
XMMATRIX P = XMMatrixPerspectiveFovLH(0.25f*XM_PI,
AspectRatio(), 1.0f, 1000.0f);
```

纵横比采用的是我们窗口的宽高比：

```
float D3DApp::AspectRatio()const
{
  return static_cast<float>(mClientWidth) / mClientHeight;
}
```

5.7　曲面细分阶段

曲面细分阶段（tessellation stages）是利用镶嵌化处理技术对网格中的三角形进行细分（subdivide），以此来增加物体表面上的三角形数量。再将这些新增的三角形偏移到适当的位置，使网格表现出更加细腻的细节（见图5.26）。

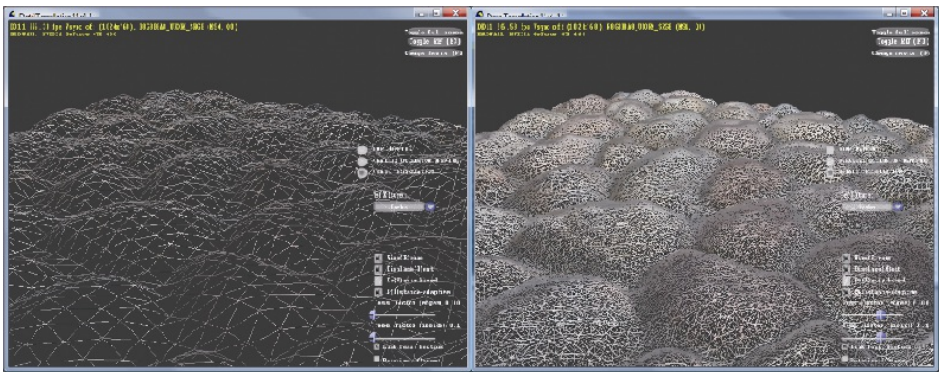


图5.26　左图展示的是原始网格，右图呈现的是经曲面细分阶段处理后的网格

使用曲面细分的优点有以下几方面。

1．我们能借此实现一种细节层次（level-of-detail，LOD）机制，使离虚拟摄像机较近的三角形经镶嵌化处理得到更加丰富的细节，而对距摄像机较远的三角形不进行任何更改。通过这种方式，即可只针对用户关注度高的部分网格增添三角形，从而提升其细节效果。

2．我们在内存中仅维护简单的低模（low-poly，低精度模型，也有译作低面多边形、低面片等）网格（低模网格是指三角形数量较少的网格，已逐渐形成一门独特画风的艺术制作手段），再根据需求为它动态

地增添额外的三角形，以此节省内存资源。

3．我们可以在处理动画和物理模拟之时采用简单的低模网格，而仅在渲染的过程中使用经镶嵌化处理的高模（high-poly，与低模对应）网格。

曲面细分是Direct3D 11中新引入的处理阶段，它们为我们提供了一种利用GPU即可对几何体进行镶嵌化处理的手段。在Direct3D 11之前，如果我们希望实现曲面细分操作，只能在CPU上实现这项任务，而且经细分后几何体必须上传回GPU中，方可进行渲染。然而，将新几何体从CPU端的内存上传至GPU显存的过程十分缓慢，而且曲面细分计算也会增加CPU的负担。由于这些原因，在Direct3D 11之前，曲面细分方法在实时渲染图像方面并没有流行开来。自Direct3D 11提供了一组相关的API起，才使得曲面细分技术完全可以在与Direct3D 11兼容的显卡中得以实现。如此一来就大大提高了曲线细分技术的魅力。曲面细分是一个可选的渲染阶段（可在用户需要之时才开启）。我们将在第14章中对此阶段进行详细的讲解。

5.8　几何着色器阶段

几何着色器（geometry shader stage，GS）是一个可选渲染阶段，由于我们从第12章起才会用到它，所以这里只对其进行简要概述。几何着色器接受的输入应当是完整的图元。例如，假设我们正在绘制三角形列表，那么向几何着色器传入的将是定义三角形的3个顶点。（注意，这3个顶点在此之前已经过了顶点着色器阶段的处理）几何着色器的主要优点是可以创建或销毁几何体。比如说，我们可以利用几何着色器将输入的图元拓展为一个或多个其他图元，抑或根据某些条件而选择不输出任何图元。顶点着色器与之相比，则不能创建顶点：它只能接受输入的单个顶点，经处理后再将该顶点输出。几何着色器的常见拿手好戏是将一个点或一条线扩展为一个四边形。

我们也可以留心观察一下图5.11中那条“流输出（stream-out）”阶段的箭头。这也就意味着，几何着色器能够为后续的绘制操作，而将顶点数据流输出至显存中的某个缓冲区之内，我们将在后续章节中对这种高级技术展开讨论。

5.9　裁剪

完全位于视锥体（viewing frustum，用户在3D空间中的可视范围（形如平截头体）亦常被称为视锥体，也有译作视平截头体、视体、视景体等）之外的几何体需要被丢弃，而处于平截头体交界以外的几何体部分也一定要接受被裁剪（clip）的操作。因此，只有在平截头体之内的物体对象才会最终保留下来。图5.27详细地演示了裁剪处理效果。

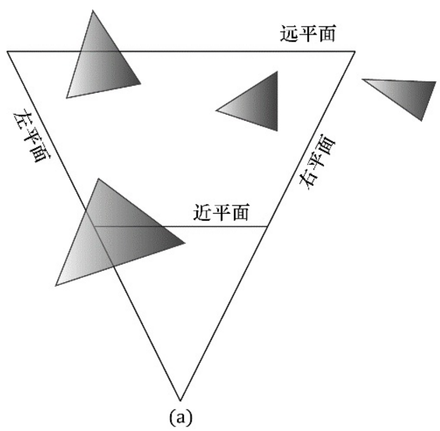


图5.27　裁剪前后对比

（a）裁剪之前　（b）裁剪之后

我们可以把平截头体看作由顶、底、左、右、近、远这6个平面所围成的空间范围。为了裁剪一个与平截头体相交的多边形，我们需要对两者相交的每个平面都逐一进行裁剪操作。当对某个存在相交的平面进行裁剪处理时（见图5.28），应保留多边形位于正半空间内的部分，并舍去其负半空间内的部分。在对与平面相交的凸多边形裁剪后，最终得到的也一定是个凸多边形。由于裁剪操作是由硬件来负责的，所以我们也就不再赘述其具体的实现细节。但是，我们在此向读者推荐一种比较

流行的裁剪方法：苏泽兰（萨瑟兰德）-霍奇曼裁剪算法（Sutherland-Hodgman clipping algorithm，前者Ivan Sutherland是图形界的奠基人，可以了解一下）[Sutherland74]。总的来说，此算法的整体思路是找到平面与多边形的所有交点，并将这些顶点按顺序组织成新的裁剪多边形。

[Blinn78]叙述了如何在4D齐次空间中进行裁剪操作（见图5.29）。在计算完透视除法之后，位于视锥体内的点就都要用规格化设备坐标$\left(\frac{x}{w},\frac{y}{w},\frac{z}{w},1\right)$来表示。而坐标中的分量范围也分别是：

$$-1 \leqslant x/w \leqslant 1$$

$$-1 \leqslant y/w \leqslant 1$$

$$0 \leqslant z/w \leqslant 1$$

在进行透视除法之前，位于齐次裁剪空间中视锥体内的4D点（x,y,z,w）将被限制在以下范围里：

$$-w \leqslant x \leqslant w$$

$$-w \leqslant y \leqslant w$$

$$0 \leqslant z \leqslant w$$

也就是说，这些点将位于下列6个4D平面所围成的空间之内：

左平面：$w = x$

右平面：$w = x$

底平面：$w = y$

顶平面：$w = y$

近平面：$z = 0$

远平面：$z = w$

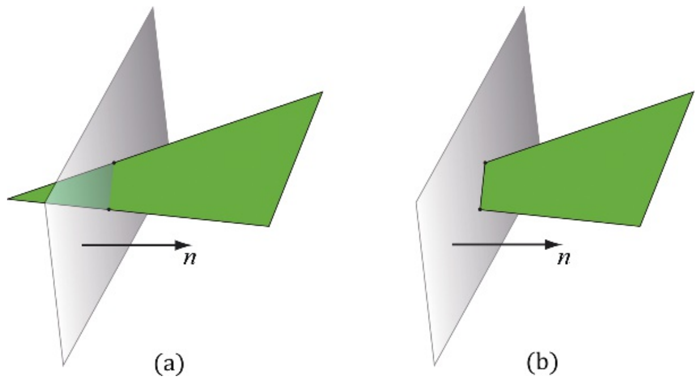


图5.28　图a中，裁剪一个与平面相交的三角形。图b中，裁剪后的三角形。可以看出经过裁剪的三角形已不再是一个三角形，而是一个四边形。因此，硬件还需要将此四边形分解为多个三角形，这对于凸多边形来讲是比较容易实现的（向量$\boldsymbol{n}$用于区分正半空间与负半空间）

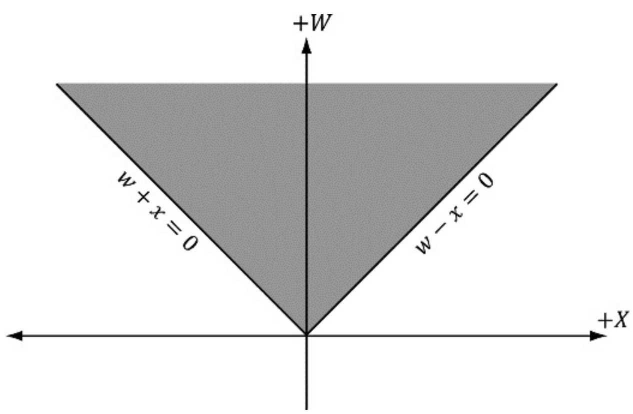


图5.29　在齐次裁剪空间中，视锥体位于xw平面内的范围

只要确定了视锥体在齐次空间内的平面方程，我们即可运用裁剪算法（例如苏泽兰-霍奇曼裁剪算法）。注意，由于可以从数学角度上将线段与平面的相交检测推广到$\mathbb{R}^4$中去，所以我们能够用4D点和4D平面

在齐次裁剪空间中进行该测试。

5.10 光栅化阶段

光栅化阶段（rasterization stage，RS，亦有将rasterization译作像素化或栅格化）的主要任务是为投影至屏幕上的3D三角形计算出对应的像素颜色。

5.10.1 视口变换

当裁剪操作完成之后，硬件会通过透视除法将物体从齐次裁剪空间变换为规格化设备坐标（NDC）。一旦物体的顶点位于NDC空间内，构成2D图像的2D顶点x、y坐标就会被变换到后台缓冲区中称为视口（viewport）的矩形里（回顾4.3.9节）。待此变换完成后，这些x、y坐标都将以像素为单位表示。通常来讲，由于z坐标常在深度缓冲技术中用作深度值，因此视口变换是不会影响此值的。即便如此，我们还是可以通过修改`D3D12_VIEWPORT`结构体中的`MinDepth`和`MaxDepth`值来做到这一点。届时，我们只需保证`MinDepth`和`MaxDepth`的取值为0～1即可。

5.10.2 背面剔除

每个三角形都有两个面，我们采用以下约定来对它们进行区分。如果组成三角形的顶点顺序为$\boldsymbol{v}_0$、$\boldsymbol{v}_1$、$\boldsymbol{v}_2$，那么，我们通过下述方法来计算此三角形的法线$\boldsymbol{n}$：

$$
\begin{aligned}
\boldsymbol{e}_0 &= \boldsymbol{v}_1 - \boldsymbol{v}_0 \\
\boldsymbol{e}_1 &= \boldsymbol{v}_2 - \boldsymbol{v}_0 \\
\boldsymbol{n} &= \frac{\boldsymbol{e}_0 \times \boldsymbol{e}_1}{\|\boldsymbol{e}_0 \times \boldsymbol{e}_1\|}
\end{aligned}
$$

法向量[11]由正面（front side）射出，另一面则为背面（back side）。图5.30详细说明了这一点。

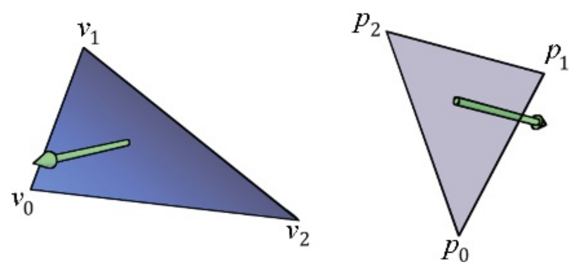


图5.30　从我们的视角来看，左侧的三角形为正面朝向，右侧的三角形为背面朝向

如果观察者看到的是三角形的正面，我们就称此三角形是正面朝向（front-facing）的；如果观察者看到的是三角形的背面，则称此三角形是背面朝向（back-facing）的。从我们的视角来看，图5.30中左侧三角形为正面朝向，右侧三角形为背面朝向。此外，根据我们的视角看来，组成左侧三角形的顶点顺序为顺时针方向，而右侧三角形的顶点顺序为逆时针方向。此现象绝非巧合：在我们选择的这种约定当中（也就是计算三角形法线的方法），根据观察者的视角看去，顶点绕序为顺时针方向的三角形为正面朝向，而顶点绕序为逆时针方向的三角形为背面朝向。

就目前我们所遇到的情况而言，大多数位于3D世界空间中的物体皆为实体对象（solid object，具有边界、表面和体积的对象，除此之外还有线框对象等。）。假设依上述约定，令构建每个物体所用都是法线

指向外侧的三角形，那么摄像机将看不到实体对象中背面朝向的三角形，这是因为正面朝向的三角形会把背面朝向的三角形遮挡起来。图5.31和图5.32分别从2D和3D的视角展示了这种现象。由于背面朝向的三角形都被正面朝向的三角形所遮挡，所以绘制它们是没有意义的。背面剔除（backface culling，也称背面消隐）就是用于将背面朝向的三角形从渲染流水线中除去的处理流程。这种操作能将待处理的三角形总量削减一半。

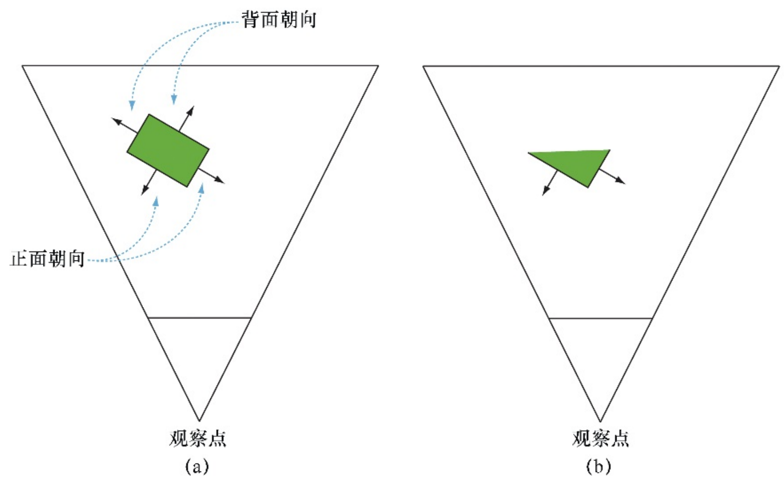


图5.31　图a中，一个由正面朝向三角形和背面朝向三角形所构成的实体对象。图b中，展示的是剔除掉背面朝向三角形的效果。注意，背面剔除技术并不会影响最终显示的图像，这是因为背面朝向的三角形都被正面朝向的三角形遮挡住了

在默认的情况下，Direct3D将以观察者的视角把顺时针绕序的三角形看作正面朝向，把逆时针绕序的三角形当作背面朝向。但是，通过对Direct3D渲染状态的设置，我们也可以将这个约定“颠倒”过来。

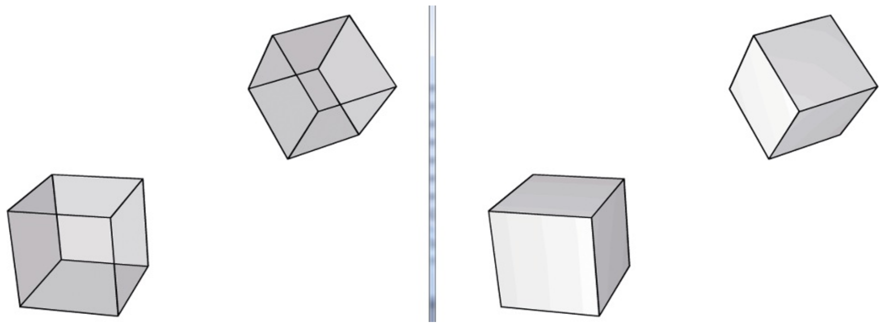


图5.32　左图中，我们以透视的方式来绘制立方体，因此能将它的6个面一览无遗右图中，我们以实体模式来绘制立方体。注意，由于3个背面朝向的面被3个正面朝向的面所遮挡，所以我们是看不到它们的——这就是说，我们可以在观察者察觉不到的情况下，从后续的处理流程中丢弃掉背面朝向的三角形

5.10.3　顶点属性插值

回顾前文可知，我们要通过指定顶点来定义三角形。除了位置信息以外，我们还能给顶点附加颜色、法向量和纹理坐标等其他属性。经过视口变换之后，我们需要为求取三角形内诸像素所附的属性而进行插值（interpolate，也有译作内插）运算。而且，除了上述顶点属性，还需对顶点的深度值进行内插，继而得到每个像素参与实现深度缓冲算法的深度值。为了得到屏幕空间（screen space，即将3D场景渲染为最终图像的2D空间）中各个顶点的插值属性，往往要通过一种名为透视校正插值（perspective correct interpolation）的方法，对3D空间中三角形的属性进行线性插值（见图5.33）。从本质上来说，插值法即利用三角形顶点的属性值计算出其内部像素的属性值。

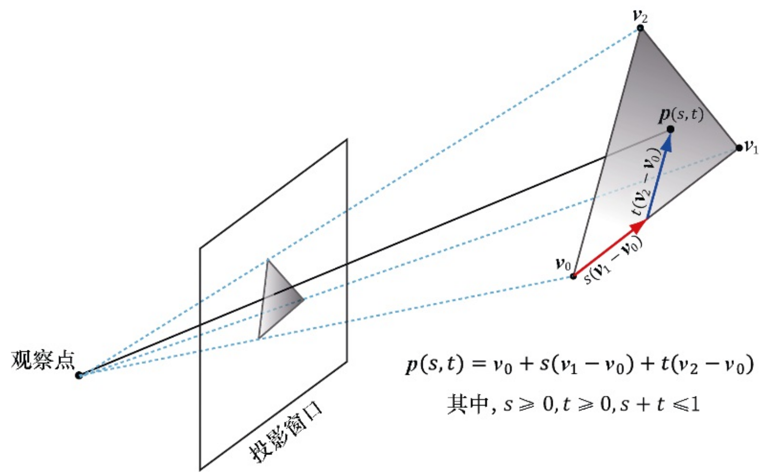


图5.33　通过对图中三角形上3个顶点的属性进行线性插值，即可求出该三角形内任意一点所具有的属性值$\boldsymbol{p}(s,t)$

我们无须考虑透视校正插值法处理像素属性的具体数学细节，因为硬件会自动完成相应的处理。对此感兴趣的读者可以从[Eberly01]中找到相应的数学推导过程。在此，仅借图5.34给出此插值法的基本思路。

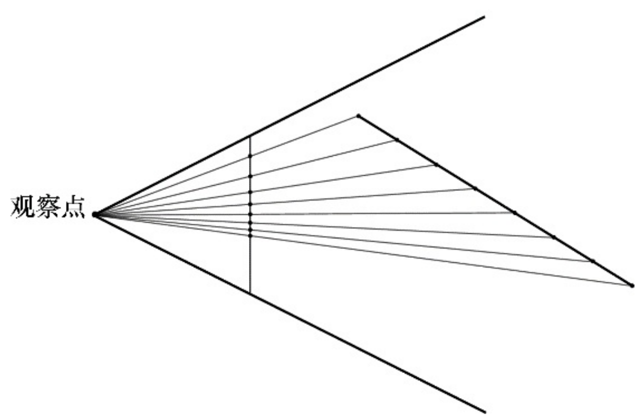


图5.34　将一条3D线段投影到投影窗口上的过程（该3D线段在屏幕空间中的投影实为一条2D线段）。可以看到，在观察空间里，与分段均匀的3D线段所对应的却可能是一条有着非均匀步长的2D线段。由此可见，如果我们在3D空间中执行的是线性插值，那么在屏幕空间里则要进行非线性插值

5.11 像素着色器阶段

我们编写的像素着色器（pixel shader，PS）是一种由GPU来执行的程序。它会针对每一个像素片段（pixel fragment，亦有译作片元）进行处理（即每处理一个像素就要执行一次像素着色器），并根据顶点的插值属性作为输入来计算出对应的像素颜色。像素着色器既可以直接返回一种单一的恒定颜色，也可以实现如逐像素光照（per-pixel lighting）、反射（reflection）以及阴影（shadow）等更为复杂的效果。

5.12　输出合并阶段

通过像素着色器生成的像素片段会被移送至渲染流水线的输出合并（Output Merger，OM）阶段。在此阶段中，一些像素片段可能会被丢弃（例如，那些未通过深度缓冲区测试或模板缓冲区测试的像素片段）。而后，剩下的像素片段将会被写入后台缓冲区中。混合（blend，也有译作融合）操作也是在此阶段实现的，此技术可令当前处理的像素与后台缓冲区中的对应像素相融合，而不仅是对后者进行完全的覆写。一些如“透明”这样的特殊效果，也是由混合技术来实现的。我们会在第10章中专门讲解这项技术。

5.13 小结

1．根据人们在真实生活中观察物体的经验，我们可以总结出一些规律。运用这些规律，我们便可以通过2D图像模拟出3D效果的场景。我们可以观察到的规律有：平行线会聚于消失点，物体的尺寸受其深度的影响（近大远小），离观察者近的物体会遮挡其后距观察者远的物体，光照与阴影的明暗对比可刻画出3D物体的实体形状和体积感，阴影还暗示了光源的位置，并反映出场景中不同物体之间的相对位置。

2．我们用三角形网格来近似地表示物体。并通过指定三角形的 3 个顶点来定义三角形。在许多网格中都存在着顶点被不同三角形所共用的现象，而索引列表则可以用于避免因重复使用顶点而复制顶点数据所带来的冗余信息。

3．我们可以通过指定红、绿、蓝三色光的强度来描述颜色。利用此三色光不同强度的相加混色（additive mixing，也称加色法），可以使我们表示出数以千万计的颜色。我们通常用归一化范围0～1来描述三色的强度，0表示没有强度，1表示最高强度，两者之间的值表示相应的中间强度。一般来说还会加入另一种名为**alpha**分量（alpha component）的颜色分量。alpha分量通常用于表示颜色的不透明度，这在混合技术中是很有用的。算上alpha分量，我们就能用4D颜色向量(r, g, b, a)来表示颜色，其中$0 \leqslant r, g, b, a \leqslant 1$。由于需要用4D向量来描述颜色数据，所以我们也就能用`XMVECTOR`类型在代码中表示颜色。而且，在使用DirectXMath向量函数进行颜色值运算时，我们还可从SIMD技术

中受益。为了能用32位数据（32-bits）来表示颜色，每一个分量都将被分配1字节；DirectXMath库提供了`XMCOLOR`结构体来存储32位颜色值。在计算颜色向量的加法、减法和标量乘法时，除了需要将分量钳制在区间[0, 1]（对32位颜色来说区间为[0, 255]）之中，其余都与普通向量的运算相同。而其他诸如“点积”“叉积”这样的向量运算，对颜色向量而言都是没有意义的。此外，符号$\otimes$表示分量式乘法，它的定义为$(c_1, c_2, c_3, c_4) \otimes (k_1, k_2, k_3, k_4) = (c_1k_1, c_2k_2, c_3k_3, c_4k_4)$。

4．给出某个3D场景的几何描述，并在此场景中设置一台具有特定位置与朝向的虚拟摄像机，那么渲染流水线（rendering pipeline）就是根据该虚拟摄像机的视角，生成能呈现在显示器中对应2D图像的这一系列完整步骤。

5．渲染流水线可以划分为输入装配（Input Assembly，IA）阶段、顶点着色器（Vertex Shader，VS）阶段、曲面细分（tessellation）阶段、几何着色器（Geometry Shader，GS）阶段、裁剪阶段、光栅化阶段（Rasterization Shage，RS）、像素着色器（Pixel Shader，PS）阶段以及输出合并（Output Merger，OM）等重要阶段。

5.14　练习

1．构建图5.35中“金字塔”的顶点列表和索引列表。

2．考虑图5.36中列出的两个几何图形。将这两种图形的顶点列表和索引列表分别合并为一个顶点列表和一个索引列表（实现的过程中要注意：将第二个索引列表追加到第一个索引列表之时，还需要对这些追加的索引值进行更新。这是因为第二个索引列表中的原索引值所引用的仍是第二个顶点列表中的顶点，而不是合并后顶点列表中的顶点）。

3．在世界坐标系中，假设一台位于(−20, 35, −50)的虚拟摄像机正对准点(10, 0, 30)进行观察，并且已知描述向上方向的向量为(0, 1, 0)。计算此摄像机的观察矩阵。

4．现给出一视锥体，其垂直视场角$\boldsymbol{\theta} = 45°$，纵横比$r$为$a = 4/3$，近平面为$n = 1$，远平面为$f = 100$。求出此视锥体对应的透视投影矩阵。

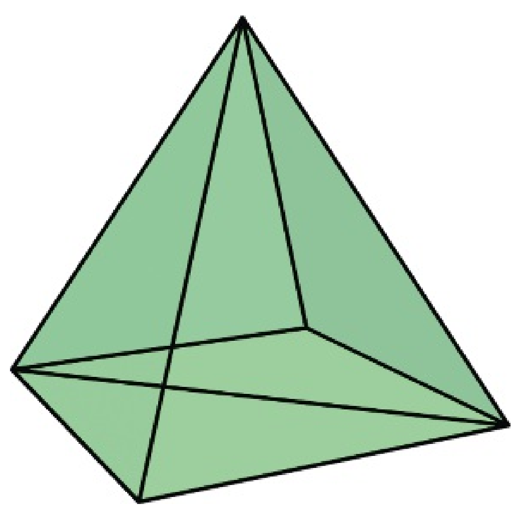


图5.35　构成四棱锥所用的多个三角形

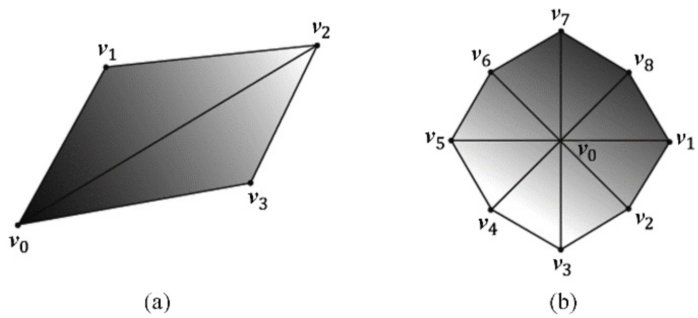


图5.36　练习2所用的几何图形

5．假设观察窗口的高度为4。求出当垂直视场角为$\boldsymbol{\theta}=60°$时，从原点到此观察窗口的距离d。

6．考虑下列透视投影矩阵：

$$\begin{bmatrix} 1.86603 & 0 & 0 & 0 \\ 0 & 3.73205 & 0 & 0 \\ 0 & 0 & 1.02564 & 1 \\ 0 & 0 & -5.12821 & 0 \end{bmatrix}$$

求出构建此矩阵所用的垂直视场角α、纵横比r以及远、近平面的值。

7．给出含有4个固定值A、B、C、D的透视投影矩阵如下：

$$\begin{bmatrix} A & 0 & 0 & 0 \\ 0 & B & 0 & 0 \\ 0 & 0 & C & 1 \\ 0 & 0 & D & 0 \end{bmatrix}$$

求出以A、B、C、D表示的构建此矩阵所用的垂直视场角α、纵横比r以及远、近平面的距离值。即求解下列方程：

（a） $A=\frac{1}{r\tan(\alpha/2)}$

（b） $B=\frac{1}{\tan(\alpha/2)}$

（c） $C=\frac{f}{f-n}$

（d） $D=\frac{-nf}{f-n}$

借助这些方程的解，我们就可得到表示本书中任意透视投影矩阵的垂直视场角α、纵横比r以及远、近平面的公式。

8．对于投影纹理算法（projective texturing algorithm，也作projective texture mapping）来说，我们在完成投影变换之后还要为之乘上一个仿射变换矩阵$\boldsymbol{T}$。验证：透视除法与乘以矩阵$\boldsymbol{T}$的运算顺序不影响计算结果。设$\boldsymbol{v}$为一个4D向量，$\boldsymbol{P}$为一个投影矩阵，$\boldsymbol{T}$为一个4×4的仿射变换矩阵，且下标w表示4D向量的w坐标，证明：

$$\left(\frac{\boldsymbol{vP}}{(\boldsymbol{vP})_w}\right)\boldsymbol{T}=\frac{(\boldsymbol{vPT})}{(\boldsymbol{vPT})_w}$$

9．证明投影矩阵的逆矩阵为：

$$\boldsymbol{P}^{-1}=\begin{bmatrix} r\tan\left(\frac{\alpha}{2}\right) & 0 & 0 & 0 \\ 0 & \tan\left(\frac{\alpha}{2}\right) & 0 & 0 \\ 0 & 0 & 0 & -\frac{f-n}{nf} \\ 0 & 0 & 1 & \frac{1}{n} \end{bmatrix}$$

10．设$[x,y,z,1]$为观察空间中的某点坐标，而且此点在NDC空间中

的坐标为$[x_{ndc}, y_{ndc}, z_{ndc}, 1]$。证明我们能通过以下方法将该点从NDC空间变换到观察空间：

$$[x_{ndc}, y_{ndc}, z_{ndc}, 1]P^{-1} = \left[\frac{x}{z}, \frac{y}{z}, 1, \frac{1}{z}\right] \xrightarrow{\text{除以}w} [x, y, z, 1]$$

解释我们为什么还需要除以w。如果是把此点从齐次裁剪空间变换到观察空间，还需除以w吗？

11．描述视锥体的另一种方法是指定其近平面处的宽和高。如若给出一个近平面为n、远平面为f，而且近平面处宽为w、高为h的视锥体，那么请证明其对应的透视投影矩阵为：

$$\boldsymbol{P} = \begin{bmatrix} \frac{2n}{w} & 0 & 0 & 0 \\ 0 & \frac{2n}{h} & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & \frac{-nf}{f-n} & 0 \end{bmatrix}$$

12．给出一个视锥体，其垂直视场角为θ，纵横比为a，近平面为n，远平面为f。求该视锥体的8个顶点坐标。

13．考虑由公式$\boldsymbol{S}_{xy}(x, y, z) = (x + zt_x, y + zt_y, z)$给出的3D剪切变换（shear transform，亦称错切变换）。变换过程详见图5.37。证明此线性变换可由以下矩阵来表示：

$$\boldsymbol{S}_{xy} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

14．思考平面$z = 1$内的3D点，即所有可由坐标$(x, y, 1)$来统一表示的

点。从上一习题中对点$(x, y, 1)$进行的剪切变换$\boldsymbol{S}_{xy}$可以看出，其效果就如同对平面$z=1$中的所有点执行了一次2D平移操作：

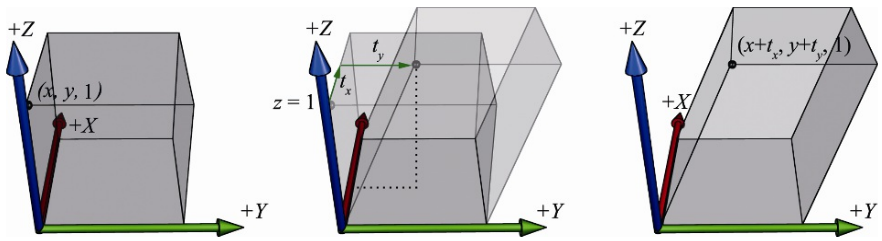


图5.37　x、y坐标随z坐标发生剪切变换。图中长方体的上表面位于平面$z=1$内。经观察可知，剪切变换对此平面上的点逐一进行了平移

$$[x,\ y,\ 1]\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ t_x & t_y & 1\end{bmatrix}=[x+t_x,\quad y+t_y,\quad 1]$$

事实上，我们也可以运用3D坐标来编写2D应用程序，但此时处理的将总是$z=1$这一平面内的2D空间。接下来，我们就可以用$\boldsymbol{S}_{xy}$在2D空间中实现平移操作。

基于上述讨论，即可推广出以下结论：

（a）正如3D空间中的平面是一个2D空间一般，4D空间中的平面实为一个3D空间。当我们用齐次坐标$(x, y, z, 1)$表示点时，其实是在处理4D平面$w=1$这一3D空间中的点。

（b）平移矩阵是4D剪切变换$\boldsymbol{S}_{xyz}(x, y, z, w)=(x+wt_x, y+wt_y, z+wt_z, w)$的矩阵表示。4D剪切变换会对平面$w=1$内的点实现平移的效果。

[1]　亦有译作渲染管道、渲染管线、绘制流水线等。有时也称之为图形流水线，即graphics pipeline。不管何种译法，希望会给读者这样一个印象：把渲染流水线想象为一个工厂里的流水线，里面有不同的加工环节（也就是渲染阶段），可以根据用户需求对每个环节灵活改造或拆卸（可编程流水线，程序员可在不同的着色器中编写自定义的函数，早期均为固定功能流水线，后加入可编程处理器予以实现。以及开启或禁用某些渲染阶段，如曲面细分阶段与几何着色器阶段等）。以此把原始材料（CPU端向GPU端提交的纹理等资源以及指令等）加工为成品出售给消费者（在GPU端，资源流经流水线里的各个阶段，经指令的调度对其进行处理，最终计算出像素的颜色，将其呈现在用户屏幕上）。事实上，渲染流水线是种模型，将3D场景变换至2D场景的处理流程抽象分离为不同的流水线阶段，供用户使用。其本质即指令从CPU端的应用程序层发送至Direct3D运行时、驱动层及至GPU端（包括二者间的通信，连接都靠PCIe接口，实质上就是围绕这种总线传递数据），资源数据在内存与显存间游走，最后是GPU内部各种引擎、缓存、命令队列等根据指令配合运作将数据转化为显示器可视信号。

[2]　此处的“实体”可以理解为“实心物体”之意，绘图模式（或模型）可划分为“实体”与“线框”两种。

[3]　已被Autodesk收入囊中。

[4]　可以尝试在彩色显示器的屏幕上点一个小水珠，并查看效果。

[5]　准确来说，alpha通道仅用于指示颜色的透明程度，alpha本身并无

透明或不透明之意，文中说的也很清楚。

[6]　DirectXMath库时有更新，所以本书的代码可能会与最新的代码有细小差异，但是功能和基本结构均保持不变。

[7]　意即构造器的输入参数为4个[0, 1]区间内的浮点分量，经过实例化后，便会将分量转换至[0, 255]区间。

[8]　在DirectX 12中，图5.13d里三角形带的实际环绕顺序为：012、132、234、354等（详见《Primitive Topologies》（bb205124）或《Triangle Strips》（bb206274））。按道理来讲，次序为偶数的三角形的顶点绕序也应遵循默认的顶点编号顺序（如第2、4个三角形的默认顶点编号顺序应为123、345），但事实上并非如此。为什么要这样做呢？作者讲到：为了使绕序保持一致，都为顺时针。而绕序又与5.10.2节中所述的剔除技术有关。（这样来看，作者似乎把偶数三角形中要调换的顶点理解错了，在DirectX中“置换”的是后两个顶点的顺序，而在OpenGL里“置换”的才是前两个顶点的顺序。）

[9]　每个图形适配器都具有特定大小的缓存（cache），刚处理过的顶点可以被临时存储在缓存当中，由于缓存的读取速度较顶点缓冲区快，因此可以利用这一点来提升软件的性能。不同硬件的缓存大小有别，因此应安排好顶点顺序，首先引用需要复用的顶点，在这些顶点仍位于缓存之中时尽快引用。

[10]　简单来讲，局部坐标系、依目标物体而建（局部）；世界坐标系是物体相对于它来定方位（全局）。

[11]　从上下文来看，本书中的法线（normal）和法向量（normal vector）指的是同一概念。

第6章　利用Direct3D绘制几何体

在第5章中，我们几乎把全部的注意力都集中在渲染流水线的概念及其相关的数学知识之上。在本章中，我们将关注配置渲染流水线的Direct3D API接口与方法、定义顶点着色器和像素着色器并将几何体提交至渲染流水线进行绘制。完成本章的学习后，我们将能以实体着色（solid coloring）模式或在线框模式（wireframe mode）下绘制一个3D立方体。

学习目标：

1．探索用于定义、存储和绘制几何体数据的Direct3D接口与方法。

2．学习编写简单的顶点着色器和像素着色器。

3．了解如何用渲染流水线状态对象来配置渲染流水线。

4．理解怎样创建常量缓冲区数据，并将其绑定到渲染流水线上。掌握根签名的用法。

6.1 顶点与输入布局

回顾5.5.1节可知，除了空间位置，Direct3D中的顶点还可以存储其他属性数据。为了构建自定义的顶点格式，我们首先要创建一个结构体来容纳选定的顶点数据。例如，下面列出了两种不同类型的顶点格式：一种由位置和颜色信息组成，另一种则由位置、法向量以及两组2D纹理坐标构成。

```
struct Vertex1
{
  XMFLOAT3 Pos;
  XMFLOAT4 Color;
};

struct Vertex2
{
  XMFLOAT3 Pos;
  XMFLOAT3 Normal;
  XMFLOAT2 Tex0;
  XMFLOAT2 Tex1;
};
```

定义了顶点结构体之后，我们还需要向Direct3D提供该顶点结构体的描述，使它了解应怎样来处理结构体中的每个成员。用户提供给Direct3D的这种描述被称为输入布局描述（input layout description），用结构体`D3D12_INPUT_LAYOUT_DESC`来表示：

```
typedef struct D3D12_INPUT_LAYOUT_DESC
{
  const D3D12_INPUT_ELEMENT_DESC *pInputElementDescs;
  UINT NumElements;
} D3D12_INPUT_LAYOUT_DESC;
```

输入布局描述实由两部分组成：即一个以D3D12_INPUT_ELEMENT_DESC元素构成的数组，以及一个表示该数组中的元素数量的整数。

D3D12_INPUT_ELEMENT_DESC数组中的元素依次描述了顶点结构体中所对应的成员。这就是说，如果某顶点结构体中有两个成员，那么与之对应的D3D12_INPUT_ELEMENT_DESC数组也将存有两个元素。D3D12_INPUT_ELEMENT_DESC结构体的定义如下：

```
typedef struct D3D12_INPUT_ELEMENT_DESC
{
  LPCSTR SemanticName;
  UINT SemanticIndex;
  DXGI_FORMAT Format;
  UINT InputSlot;
  UINT AlignedByteOffset;
  D3D12_INPUT_CLASSIFICATION InputSlotClass;
  UINT InstanceDataStepRate;
} D3D12_INPUT_ELEMENT_DESC;
```

1. SemanticName：一个与元素相关联的特定字符串，我们称之为语义（semantic），它传达了元素的预期用途。该参数可以是任意合法的语义名。通过语义即可将顶点结构体（图6.1中的struct Vertex）中的元素与顶点着色器输入签名[1]（vertex shader input signature，即图6.1 中的VertexOut VS的众参数）中的元素一一映射起来，如图6.1所示。

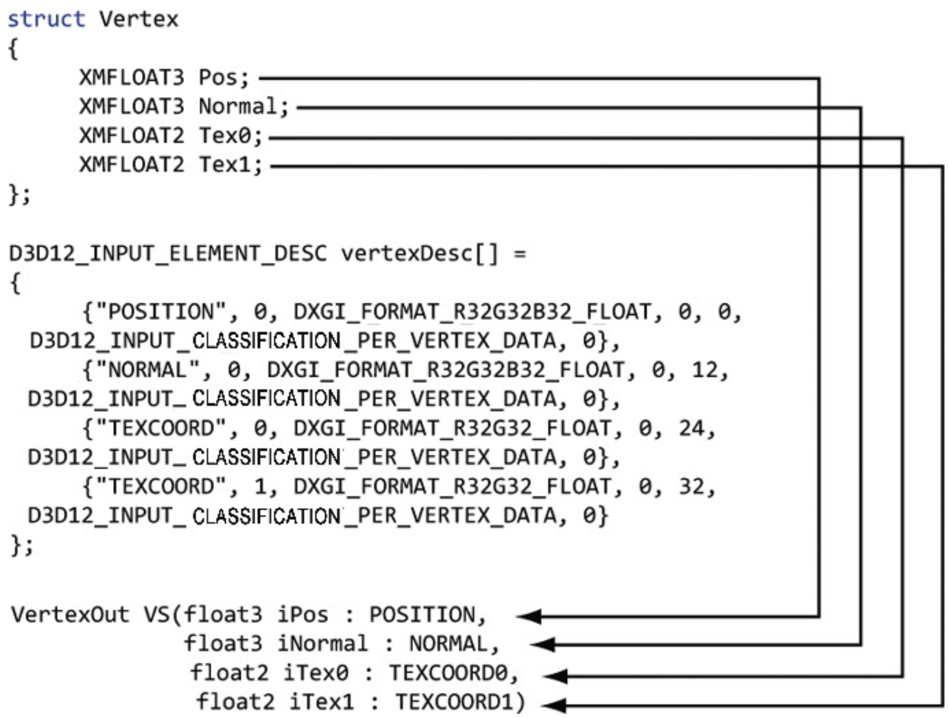


图6.1　D3D12_INPUT_ELEMENT_DESC数组中的元素与顶点结构体中的成员一一对应，而语义名与索引则为顶点元素映射到顶点着色器中对应的参数提供了途径

2．SemanticIndex：附加到语义上的索引。此成员的设计动机可从图6.1中看出。例如，顶点结构体中的纹理坐标可能不止一组，而仅在语义名尾部添加一个索引，即可在不引入新语义名的情况下区分出这两组不同的纹理坐标。在着色器代码中，未标明索引的语义将默认其索引值为0，也就是说，图6.1中的POSITION与POSITION0等价。

3．Format：在Direct3D中，要通过枚举类型DXGI_FORMAT中的成员来指定顶点元素的格式（即数据类型）。下面是一些常用的格式。

```
DXGI_FORMAT_R32_FLOAT       // 1D 32位浮点标量
DXGI_FORMAT_R32G32_FLOAT     // 2D 32位浮点向量
DXGI_FORMAT_R32G32B32_FLOAT   // 3D 32位浮点向量
DXGI_FORMAT_R32G32B32A32_FLOAT // 4D 32位浮点向量

DXGI_FORMAT_R8_UINT      // 1D 8位无符号整型标量
```

```
  DXGI_FORMAT_R16G16_SINT    // 2D 16位有符号整型向量
  DXGI_FORMAT_R32G32B32_UINT // 3D 32位无符号整型向量
  DXGI_FORMAT_R8G8B8A8_SINT  // 4D 8位有符号整型向量
  DXGI_FORMAT_R8G8B8A8_UINT  // 4D 8位无符号整型向量
```

4．`InputSlot`：指定传递元素所用的输入槽（input slot index）索引。Direct3D共支持16个输入槽（索引值为0～15），可以通过它们来向输入装配阶段传递顶点数据。目前我们只会用到输入槽0（即所有的顶点元素都来自同一个输入槽），但在本章的习题2中将会涉及多输入槽的编程实践。

5．`AlignedByteOffset`：在特定输入槽中，从C++顶点结构体的首地址到其中某点元素起始地址的偏移量（用字节表示）。例如，在下列顶点结构体中，元素`Pos`的偏移量为0字节，因为它的起始地址与顶点结构体的首地址一致；元素`Normal`的偏移量为12字节，因为跳过`Pos`所占用的字节数才能找到`Normal`的起始地址；元素`Tex0`的偏移量为24字节，因为跨过`Pos`和`Normal`的总字节数才可获取`Tex0`的起始地址。同理，元素`Tex1`的偏移量为32字节，只有跳过`Pos`、`Normal`和`Tex0`这3个元素的总字节数方可寻得`Tex1`的起始地址。

```
struct Vertex2
{
  XMFLOAT3 Pos;   // 偏移量为0字节
  XMFLOAT3 Normal; // 偏移量为12字节
  XMFLOAT2 Tex0;  // 偏移量为24字节
  XMFLOAT2 Tex1;  // 偏移量为32字节
};
```

6．`InputSlotClass`：我们暂且把此参数指定为`D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA`[2]。而另一选项

（D3D12_INPUT_CLASSIFICATION_PER_INSTANCE_DATA[3]）则用于实现实例化（instancing）这种高级技术。

7. InstanceDataStepRate：目前仅将此值指定为0。若要采用实例化这种高级技术，则将此参数设为1。

就前面Vertex1和Vertex2这两个顶点结构体的例子来说，其相应的输入布局描述为：

```
D3D12_INPUT_ELEMENT_DESC desc1[] =
{
  {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
  {"COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0}
};

D3D12_INPUT_ELEMENT_DESC desc2[] =
{
  {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
  {"NORMAL",  0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
  {"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT,  0, 24,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0}
  {"TEXCOORD", 1, DXGI_FORMAT_R32G32_FLOAT,  0, 32,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0}
};
```

6.2　顶点缓冲区

为了使GPU可以访问顶点数组，就需要把它们放置在称为缓冲区（buffer）的GPU资源（ID3D12Resource）里。我们把存储顶点的缓冲区叫作顶点缓冲区（vertex buffer）。缓冲区的结构比纹理更为简单：既非多维资源，也不支持mipmap、过滤器以及多重采样等技术。当需要向GPU提供如顶点这类数据元素所构成的数组时，我们便会使用缓冲区。

就像在4.3.8节中所做的那样，我们先通过填写D3D12_RESOURCE_DESC结构体来描述缓冲区资源，接着再调用ID3D12Device::CreateCommittedResource方法去创建ID3D12Resource对象。至于D3D12_RESOURCE_DESC结构体中所有成员的介绍，可参考4.3.8节。Direct3D 12提供了一个C++包装类CD3DX12_RESOURCE_DESC，它派生自D3D12_RESOURCE_DESC结构体，并附有多种便于使用的构造函数以及方法。特别是它提供的下列方法，一种简化缓冲区描述过程的D3D12_RESOURCE_DESC的构造函数：

```
static inline CD3DX12_RESOURCE_DESC Buffer(
    UINT64 width,
    D3D12_RESOURCE_FLAGS flags = D3D12_RESOURCE_FLAG_NONE,
    UINT64 alignment = 0 )
{
  return CD3DX12_RESOURCE_DESC( D3D12_RESOURCE_DIMENSION_BUFFER,
    alignment, width, 1, 1, 1,
    DXGI_FORMAT_UNKNOWN, 1, 0,
    D3D12_TEXTURE_LAYOUT_ROW_MAJOR, flags );
}
```

对于缓冲区而言，函数代码中的width即表示缓冲区中所占的字节数。例如，若缓冲区存储了64个float类型的数据，那么width的值即为64*sizeof(float)。

注意 Note

除此之外，CD3DX12_RESOURCE_DESC类还提供了以下构建D3D12_RESOURCE_DESC结构体的简便方法，用于描述纹理资源及其可供查询的相关信息：

1. CD3DX12_RESOURCE_DESC::Tex1D

2. CD3DX12_RESOURCE_DESC::Tex2D

3. CD3DX12_RESOURCE_DESC::Tex3D

注意 Note

我们在第4章中曾提到深度/模板缓冲区，它是一种以ID3D12Resource对象表示的2D纹理。在Direct3D 12中，所有的资源均用ID3D12Resource接口表示。相比之下，Direct3D 11则采用如ID3D11Buffer与ID3D11Texture2D等多种不同的接口来表示各种不同

的资源。而且，在Direct3D 12中，资源的类型由D3D12_RESOURCE_DESC::D3D12_RESOURCE_DIMENSION字段来加以区分。例如，缓冲区用D3D12_RESOURCE_DIMENSION_BUFFER类型表示，而2D纹理则以D3D12_RESOURCE_DIMENSION_TEXTURE2D类型表示。

对于静态几何体（static geometry，即每一帧都不会发生改变的几何体）而言，我们会将其顶点缓冲区置于默认堆（D3D12_HEAP_TYPE_DEFAULT）中来优化性能。一般说来，游戏中的大多数几何体（如树木、建筑物、地形和动画角色）都是如此处理。在这种情况下，顶点缓冲区初始化完毕后，只有GPU需要从其中读取数据来绘制几何体，所以使用默认堆是很明智的做法。然而，如果CPU不能向默认堆中的顶点缓冲区写入数据，那么我们该如何初始化此顶点缓冲区呢？

因此，除了创建顶点缓冲区资源本身之外，我们还需用D3D12_HEAP_TYPE_UPLOAD这种堆类型来创建一个处于中介位置的上传缓冲区（upload buffer）资源。在4.3.8节里，我们就是通过把资源提交至上传堆，才得以将数据从CPU复制到GPU显存中。在创建了上传缓冲区之后，我们就可以将顶点数据从系统内存复制到上传缓冲区，而后再把顶点数据从上传缓冲区复制到真正的顶点缓冲区中。

由于我们需要利用作为中介的上传缓冲区来初始化默认缓冲区（即

用堆类型D3D12_HEAP_TYPE_DEFAULT创建的缓冲区）中的数据，因此，我们就在d3dUtil.h/.cpp文件中构建了下列工具函数，以避免在每次使用默认缓冲区时再做这些重复的工作。

```
Microsoft::WRL::ComPtr d3dUtil::CreateDefaultBuffer(
  ID3D12Device* device,
  ID3D12GraphicsCommandList* cmdList,
  const void* initData,
  UINT64 byteSize,
  Microsoft::WRL::ComPtr& uploadBuffer)
{
  ComPtr defaultBuffer;

  // 创建实际的默认缓冲区资源
  ThrowIfFailed(device->CreateCommittedResource(
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
    D3D12_HEAP_FLAG_NONE,
    &CD3DX12_RESOURCE_DESC::Buffer(byteSize),
       D3D12_RESOURCE_STATE_COMMON,
    nullptr,
    IID_PPV_ARGS(defaultBuffer.GetAddressOf())));

  // 为了将CPU端内存中的数据复制到默认缓冲区，我们还需要创建一个处于中介位置的上
传堆
  ThrowIfFailed(device->CreateCommittedResource(
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
       D3D12_HEAP_FLAG_NONE,
    &CD3DX12_RESOURCE_DESC::Buffer(byteSize),
       D3D12_RESOURCE_STATE_GENERIC_READ,
    nullptr,
    IID_PPV_ARGS(uploadBuffer.GetAddressOf())));

  // 描述我们希望复制到默认缓冲区中的数据
  D3D12_SUBRESOURCE_DATA subResourceData = {};
  subResourceData.pData = initData;
  subResourceData.RowPitch = byteSize;
  subResourceData.SlicePitch = subResourceData.RowPitch;

  // 将数据复制到默认缓冲区资源的流程
  // UpdateSubresources辅助函数会先将数据从CPU端的内存中复制到位于中介位置的上
传堆里接着，
  // 再通过调用ID3D12CommandList::CopySubresourceRegion函数，把上传堆内的数据
复制到
  // mBuffer中[4]
```

```
  cmdList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER::Transition(defaultBuffer.Get(),
    D3D12_RESOURCE_STATE_COMMON,
    D3D12_RESOURCE_STATE_COPY_DEST));
  UpdateSubresources(cmdList,
    defaultBuffer.Get(), uploadBuffer.Get(),
    0, 0, 1, &subResourceData);
  cmdList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER::Transition(defaultBuffer.Get(),
    D3D12_RESOURCE_STATE_COPY_DEST,
    D3D12_RESOURCE_STATE_GENERIC_READ));

  // 注意：在调用上述函数后，必须保证uploadBuffer依然存在，而不能对它立即进行销毁。这是因为
  // 命令列表中的复制操作可能尚未执行。待调用者得知复制完成的消息后，方可释放uploadBuffer
  return defaultBuffer;
}
```

D3D12_SUBRESOURCE_DATA结构体的定义为：

```
typedef struct D3D12_SUBRESOURCE_DATA
{
  const void *pData;
  LONG_PTR RowPitch;
  LONG_PTR SlicePitch;
} D3D12_SUBRESOURCE_DATA;
```

1. pData：指向某个系统内存块的指针，其中有初始化缓冲区所用的数据。如果欲初始化的缓冲区能够存储n个顶点数据，则该系统内存块必定可容纳至少n个顶点数据，以此来初始化整个缓冲区。

2. RowPitch：对于缓冲区而言，此参数为欲复制数据的字节数。

3. SlicePitch：对于缓冲区而言，此参数亦为欲复制数据的字节数。

下面的代码演示了此类将如何创建存有立方体8个顶点的默认缓冲

区，并为其中的每个顶点都分别赋予了不同的颜色。

```
Vertex vertices[] =
{
  { XMFLOAT3(-1.0f, -1.0f, -1.0f), XMFLOAT4(Colors::White) },
  { XMFLOAT3(-1.0f, +1.0f, -1.0f), XMFLOAT4(Colors::Black) },
  { XMFLOAT3(+1.0f, +1.0f, -1.0f), XMFLOAT4(Colors::Red) },
  { XMFLOAT3(+1.0f, -1.0f, -1.0f), XMFLOAT4(Colors::Green) },
  { XMFLOAT3(-1.0f, -1.0f, +1.0f), XMFLOAT4(Colors::Blue) },
  { XMFLOAT3(-1.0f, +1.0f, +1.0f), XMFLOAT4(Colors::Yellow) },
  { XMFLOAT3(+1.0f, +1.0f, +1.0f), XMFLOAT4(Colors::Cyan) },
  { XMFLOAT3(+1.0f, -1.0f, +1.0f), XMFLOAT4(Colors::Magenta) }
};

const UINT64 vbByteSize = 8 * sizeof(Vertex);

ComPtr<ID3D12Resource> VertexBufferGPU = nullptr;
ComPtr<ID3D12Resource> VertexBufferUploader = nullptr;
VertexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
  mCommandList.Get(), vertices, vbByteSize, VertexBufferUploader);
```

代码中的Vertex类型（即顶点的坐标及其颜色）的定义如下。

```
struct Vertex
{
  XMFLOAT3 Pos;
  XMFLOAT4 Color;
};
```

为了将顶点缓冲区绑定到渲染流水线上，我们需要给这种资源创建一个顶点缓冲区视图（vertex buffer view）。与RTV（render target view，渲染目标视图）不同的是，我们无须为顶点缓冲区视图创建描述符堆。而且，顶点缓冲区视图是由D3D12_VERTEX_BUFFER_VIEW[5]结构体来表示。

```
typedef struct D3D12_VERTEX_BUFFER_VIEW
{
  D3D12_GPU_VIRTUAL_ADDRESS BufferLocation;
```

```
  UINT SizeInBytes;
  UINT StrideInBytes;
}   D3D12_VERTEX_BUFFER_VIEW;
```

1．BufferLocation：待创建视图的顶点缓冲区资源虚拟地址。我们可以通过ID3D12Resource::GetGPUVirtualAddress方法来获得此地址。

2．SizeInBytes：待创建视图的顶点缓冲区大小（用字节表示）。

3．StrideInBytes：每个顶点元素所占用的字节数。

在顶点缓冲区及其对应视图创建完成后，便可以将它与渲染流水线上的一个输入槽（input slot）相绑定。这样一来，我们就能向流水线中的输入装配器阶段传递顶点数据了。此操作可以通过下列方法来实现。

```
void ID3D12GraphicsCommandList::IASetVertexBuffers(
  UINT StartSlot,
  UINT NumView,
  const D3D12_VERTEX_BUFFER_VIEW *pViews);
```

1．StartSlot：在绑定多个顶点缓冲区时，所用的起始输入槽（若仅有一个顶点缓冲区，则将其绑定至此槽）。输入槽共有16个，索引为0～15。

2．NumViews：将要与输入槽绑定的顶点缓冲区数量（即视图数组pViews中视图的数量）。如果起始输入槽StartSlot的索引值为k，且我们要绑定n个顶点缓冲区，那么这些缓冲区将依次与输入槽

$I_k, I_{k+1}, \ldots, I_{k+n-1}$相绑定。

3. pViews：指向顶点缓冲区视图数组中第一个元素的指针。

下面是该函数的一个调用示例。

```
D3D12_VERTEX_BUFFER_VIEW vbv;
vbv.BufferLocation = VertexBufferGPU->GetGPUVirtualAddress();
vbv.StrideInBytes = sizeof(Vertex);
vbv.SizeInBytes = 8 * sizeof(Vertex);

D3D12_VERTEX_BUFFER_VIEW vertexBuffers[1] = { vbv };
mCommandList->IASetVertexBuffers(0, 1, vertexBuffers);
```

由于IASetVertexBuffers方法会将顶点缓冲区数组中的元素设置到不同的输入槽上去，所以使它看起来似乎有些复杂。但是，在我们的示例中实际只会使用一个输入槽。而在章末的一道习题中，我们将体验到运用两个输入槽进行绘制的过程。

我们若不对顶点缓冲区进行任何修改，它就将一直被绑定于所在的输入槽上。所以，如果使用多个顶点缓冲区，那么就可以按以下流程来构建代码。

```
ID3D12Resource* mVB1; // 存储Vertex1类型的顶点
ID3D12Resource* mVB2; // 存储Vertex2类型的顶点

D3D12_VERTEX_BUFFER_VIEW mVBView1; // mVB1的视图
D3D12_VERTEX_BUFFER_VIEW mVBView2; // mVB2的视图

/*……创建顶点缓冲区及其视图……*/

mCommandList->IASetVertexBuffers(0, 1, &mVBView1);
/* ……使用顶点缓冲区1来绘制物体…… */

mCommandList->IASetVertexBuffers(0, 1, &mVBView2);
/* ……使用顶点缓冲区2来绘制物体…… */
```

将顶点缓冲区设置到输入槽上并不会对其执行实际的绘制操作，而是仅为顶点数据送至渲染流水线做好准备而已。这最后一步才是通过ID3D12GraphicsCommandList::DrawInstanced方法[6]真正地绘制顶点。

```
void ID3D12GraphicsCommandList::DrawInstanced (
  UINT VertexCountPerInstance,
  UINT InstanceCount,
  UINT StartVertexLocation,
  UINT StartInstanceLocation);
```

1. VertexCountPerInstance：每个实例要绘制的顶点数量。

2. InstanceCount：用于实现一种被称作实例化（instancing）的高级技术。就目前来说，我们只绘制一个实例，因而将此参数设置为1。

3. StartVertexLocation：指定顶点缓冲区内第一个被绘制顶点的索引（该索引值以0为基准）。

4. StartInstanceLocation：用于实现一种被称作实例化的高级技术，暂时只需将其设置为0。

VertexCountPerInstance和StartVertexLocation两个参数定义了顶点缓冲区中将要被绘制的一组连续顶点，如图6.2所示。

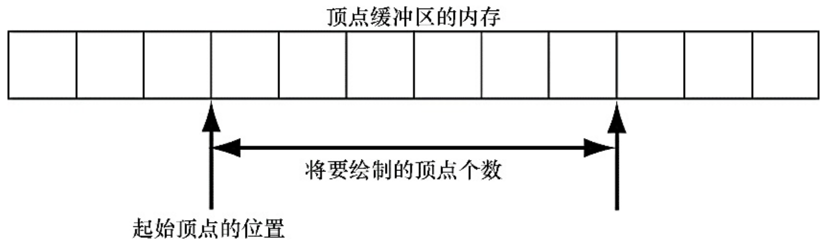


图6.2　StartVertexLocation参数指定了顶点缓冲区中第一个被绘制顶点的索引（此索引从0开始计），VertexCountPerInstance指定了欲绘制顶点的个数

既然DrawInstanced方法没有指定顶点被定义为何种图元，那么，它们应该被绘制为点、线列表还是三角形列表呢？回顾5.5.2节可知，图元拓扑状态实由ID3D12GraphicsCommandList::IASetPrimitiveTopology方法来设置。下面给出一个相关的调用示例：

```
cmdList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
```

6.3 索引和索引缓冲区

与顶点相似，为了使GPU可以访问索引数组，就需要将它们放置于GPU的缓冲区资源（ID3D12Resource）内。我们称存储索引的缓冲区为索引缓冲区（index buffer）。由于本书所采用的d3dUtil::CreateDefaultBuffer函数是通过void*类型作为参数引入泛型数据，这就意味着我们也可以用此函数来创建索引缓冲区（或任意类型的默认缓冲区）。

为了使索引缓冲区与渲染流水线绑定，我们需要给索引缓冲区资源创建一个索引缓冲区视图（index buffer view）。如同顶点缓冲区视图一样，我们也无须为索引缓冲区视图创建描述符堆。但索引缓冲区视图要由结构体D3D12_INDEX_BUFFER_VIEW来表示。

```
typedef struct D3D12_INDEX_BUFFER_VIEW
{
  D3D12_GPU_VIRTUAL_ADDRESS BufferLocation;
  UINT SizeInBytes;
  DXGI_FORMAT Format;
} D3D12_INDEX_BUFFER_VIEW;
```

1. BufferLocation：待创建视图的索引缓冲区资源虚拟地址。我们可以通过调用ID3D12Resource::GetGPUVirtualAddress方法来获取此地址。

2. SizeInBytes：待创建视图的索引缓冲区大小（以字节表示）。

3. Format：索引的格式必须为表示16位索引的DXGI_FORMAT_R16_UINT类型，或表示32位索引的DXGI_FORMAT_R32_UINT类型。16位的索引可以减少内存和带宽的占用，但如果索引值范围超过了16位数据的表达范围，则也只能采用32位索引了。

与顶点缓冲区相似（也包括其他的Direct3D资源在内），在使用之前，我们需要先将它们绑定到渲染流水线上。通过ID3D12GraphicsCommandList::IASetIndexBuffer方法即可将索引缓冲区绑定到输入装配器阶段。下面的代码演示了怎样创建一个索引缓冲区来定义构成立方体的三角形，以及为该索引缓冲区创建视图并将它绑定到渲染流水线：

```
std::uint16_t indices[] = {
  // 立方体前表面
  0, 1, 2,
  0, 2, 3,

  // 立方体后表面
  4, 6, 5,
  4, 7, 6,

  // 立方体左表面
  4, 5, 1,
  4, 1, 0,

  // 立方体右表面
  3, 2, 6,
  3, 6, 7,

  // 立方体上表面
  1, 5, 6,
  1, 6, 2,

  // 立方体下表面
  4, 0, 3,
```

```
  4, 3, 7
};

const UINT ibByteSize = 36 * sizeof(std::uint16_t);

ComPtr<ID3D12Resource> IndexBufferGPU = nullptr;
ComPtr<ID3D12Resource> IndexBufferUploader = nullptr;
IndexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
  mCommandList.Get(), indices, ibByteSize, IndexBufferUploader);

D3D12_INDEX_BUFFER_VIEW ibv;
ibv.BufferLocation = IndexBufferGPU->GetGPUVirtualAddress();
ibv.Format = DXGI_FORMAT_R16_UINT;
ibv.SizeInBytes = ibByteSize;

mCommandList->IASetIndexBuffer(&ibv);
```

最后需要注意的是，在使用索引的时候，我们一定要用ID3D12GraphicsCommandList::DrawIndexedInstanced方法代替DrawInstanced方法进行绘制。

```
void ID3D12GraphicsCommandList::DrawIndexedInstanced(
  UINT IndexCountPerInstance,
  UINT InstanceCount,
  UINT StartIndexLocation,
  INT BaseVertexLocation,
  UINT StartInstanceLocation);
```

1. IndexCountPerInstance：每个实例将要绘制的索引数量。

2. InstanceCount：用于实现一种被称作实例化的高级技术。就目前而言，我们只绘制一个实例，因而将此值设置为1。

3. StartIndexLocation：指向索引缓冲区中的某个元素，将其标记为欲读取的起始索引。

4. BaseVertexLocation：在本次绘制调用读取顶点之前，要为

每个索引都加上此整数值。

5. StartInstanceLocation：用于实现一种被称为实例化的高级技术，暂将其设置为0。

为了理解这些参数，让我们来思考这样一个情景：假设有3个欲绘制的物体，一个球体、一个立方体以及一个圆柱体。首先，每个物体都有自己的顶点缓冲区以及索引缓冲区。而每个局部索引缓冲区中的索引，又都引用的是各自的局部顶点缓冲区。现在，我们把这3个物体的顶点和索引分别连接为全局顶点缓冲区和全局索引缓冲区，如图6.3所示。（在顶点缓冲区和索引缓冲区合并的过程中，调用API时可能会产生一些开销，但这基本上不会成为程序的瓶颈。出于性能的原因，当应用中有一些可以轻易合并的零散顶点缓冲区和索引缓冲区时，便值得照这样试一试）待合并完成后，索引缓冲区里的元素就不能正确地引用对应的顶点数据了，这是因为它们存储的索引值是相对于物体各自的局部顶点缓冲区而言，而非全局的顶点缓冲区。所以，还需要对索引值进行重新计算，使之可以正确地引用全局顶点缓冲区中的顶点数据。假设原始的立方体索引是根据下述立方体顶点编号来计算的：

```
0, 1, ..., numBoxVertices-1
```

但是在顶点缓冲区与索引缓冲区一一合并之后，该立方体的索引编号将依次变为：

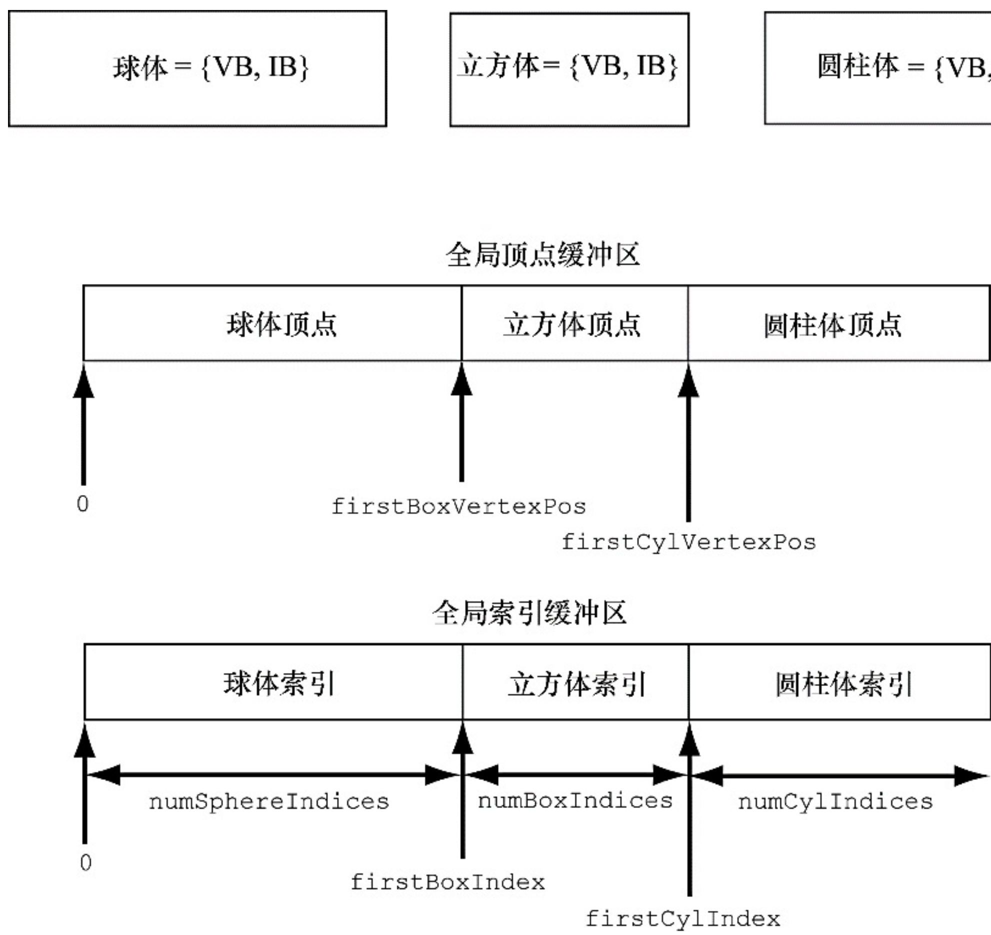


图6.3　将几个顶点缓冲区合并为一个大的顶点缓冲区，再把对应的若干索引缓冲区合并为一个大的索引缓冲区

```
firstBoxVertexPos,
firstBoxVertexPos+1,
...,
firstBoxVertexPos+numBoxVertices-1
```

因此，为了更新索引，我们需要为每个立方体的原索引加上firstBoxVertexPos（缓冲区合并后立方体第一个顶点的索引值）。类似地，我们也需要给每个圆柱体的原索引加上firstCylVertexPos（缓冲区合并后圆柱体第一个顶点的索引值）。注意，球体的索引是不需要改变的（因为球体第一个顶点的位置

始终为0，在合并后的全局索引缓冲区中也未曾改变）。我们将每个物体的第一个顶点相对于全局顶点缓冲区的位置叫作它的基准顶点地址（base vertex location）。通常来讲，一个物体的新索引是通过原始索引加上它的基准顶点地址来获取的。事实上，我们不必亲自计算新的索引值：通过向`DrawIndexedInstanced`函数的第4个参数传递基准顶点地址，即可让Direct3D去执行相关计算工作。

接下来，我们再通过下列3次绘制调用来依次绘制球体、立方体和圆柱体：

```
mCmdList->DrawIndexedInstanced(
  numSphereIndices, 1, 0, 0, 0);
mCmdList->DrawIndexedInstanced(
  numBoxIndices, 1, firstBoxIndex, firstBoxVertexPos, 0);
mCmdList->DrawIndexedInstanced(
  numCylIndices, 1, firstCylIndex, firstCylVertexPos, 0);
```

在第7章的“Shapes”示例项目中将采用这项绘制技术。

6.4　顶点着色器示例

以下代码实现的是一个简单的顶点着色器（vertex shader，回顾5.6节）：

```
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorldViewProj;
};

void VS(float3 iPosL : POSITION,
    float4 iColor : COLOR,
    out float4 oPosH : SV_POSITION,
    out float4 oColor : COLOR)
{
  // 把顶点变换到齐次裁剪空间
  oPosH = mul(float4(iPosL, 1.0f), gWorldViewProj);

  // 直接将顶点的颜色信息传至像素着色器
  oColor = iColor;
}
```

在Direct3D中，编写着色器的语言为高级着色语言（High Level Shading Language，HLSL）[7]，其语法与C++十分相似，这使得它较易于学习。附录B提供了一份简明的HLSL参考资料。我们将结合实例来学习HLSL和着色器的编写。也就是说，随着本书内容的推进，我们将为了实现手头的样例而逐步介绍与HLSL有关的新概念。一般情况下，着色器通常在以.hlsl为扩展名的文本文件中编写。

顶点着色器就是上例中名为VS的函数。值得注意的是，我们可以给顶点着色器起任意合法的函数名。上述顶点着色器共有4个参数，前两个为输入参数，后两个为输出参数（通过关键字out来表示）。HLSL

没有引用（reference）和指针（pointer）的概念，所以需要借助结构体或多个输出参数才能够从函数中返回多个数值。而且，在HLSL中，所有的函数都是内联（inline）函数。

前两个输入参数分别对应于绘制立方体所自定义顶点结构体中的两个数据成员，也构成了顶点着色器的输入签名（input signature）。参数语义“:POSITION”和“:COLOR”用于将顶点结构体中的元素映射到顶点着色器的相应输入参数，如图6.4所示。

输出参数也附有各自的语义（“:SV_POSITION”和“:COLOR”），并以此作为纽带，将顶点着色器的输出参数映射到下个处理阶段（几何着色器或像素着色器）中所对应的输入参数。注意，SV_POSITION语义比较特殊（SV代表系统值，即system value），它所修饰的顶点着色器输出元素存有齐次裁剪空间中的顶点位置信息。因此，我们必须为输出位置信息的参数附上SV_POSITION语义，使GPU可以在进行例如裁剪、深度测试和光栅化等处理之时，借此实现其他属性所无法介入的有关运算。值得注意的是，对于任何不具有系统值的输出参数而言，我们都可以根据需求以合法的语义名修饰它[8]。

```
struct Vertex
{
      XMFLOAT3 Pos;
      XMFLOAT4 Color;
};

D3D12_INPUT_ELEMENT_DESC vertexDesc[] =
{
      {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
   D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
      {"COLOR",    0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
   D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0}
};

void VS(float3 iPosL : POSITION,
        float4 iColor : COLOR,
        out float4 oPosH : SV_POSITION,
        out float4 oColor : COLOR)
{
      // Transform to homogeneous clip space.
      oPosH = mul(float4(iPosL, 1.0f), gWorldViewProj);

      // Just pass vertex color into the pixel shader.
      oColor = iColor;
}
```

图6.4　通过D3D12_INPUT_ELEMENT_DESC数组为每个顶点元素都指定了与之关联的语义。顶点着色器中的每个参数也各附有一个语义，该语义用于使顶点元素与顶点着色器参数逐一匹配

上述顶点着色器函数的第一行代码是通过将顶点与4×4矩阵gWorldViewProj相乘，使其坐标由局部空间变换到齐次裁剪空间：

```
// 将顶点坐标变换到齐次裁剪空间
oPosH = mul(float4(iPosL, 1.0f), gWorldViewProj);
```

借助构造语法float4(iPosL, 1.0f)即可构建一个等价于float4(iPosL.x, iPosL.y, iPosL.z, 1.0f)的4D向量。我们知道，顶点的位置是一个点而非向量，所以将向量的第4个分量设置为1（$w = 1$）。并以float2与float3类型分别表示2D和3D向量。矩阵变量gWorldViewProj存于常量缓冲区（constant buffer）内，我们会在6.6节中对它进行相关讨论。内置函数（built-in function,也译作内建函数、内部函数等）mul则用于计算向量与矩阵之间的乘法。顺便提一下，mul函数可以根据不同规模的矩阵乘法而重载。例如，我们可以用mul函数

进行两个4×4矩阵的乘法、两个3×3矩阵的乘法或者一个1×3向量与3×3矩阵的乘法等。着色器函数的最后一行代码把输入的颜色直接复制给输出参数，继而将该颜色传递到渲染流水线的下个阶段：

```
oColor = iColor;
```

我们可以把函数的返回类型和输入签名替换为结构体（从而取代过长的参数列表），即将以上顶点着色器改写为另一种等价实现：

```
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorldViewProj;
};

struct VertexIn
{
  float3 PosL : POSITION;
  float4 Color : COLOR;
};

struct VertexOut
{
  float4 PosH : SV_POSITION;
  float4 Color : COLOR;
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout;

  // 将顶点变换到齐次裁剪空间
  vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);

  // 直接将顶点颜色传至像素着色器
  vout.Color = vin.Color;

  return vout;
}
```

注意 Note

如果没有使用几何着色器（我们会在第12章中介绍这种着色器），那么顶点着色器必须用SV_POSITION语义来输出顶点在齐次裁剪空间中的位置，因为（在没有使用几何着色器的情况下）执行完顶点着色器之后，硬件期望获取顶点位于齐次裁剪空间之中的坐标。如果使用了几何着色器，则可以把输出顶点在齐次裁剪空间中位置的工作交给它来处理。

注意 Note

在顶点着色器（或几何着色器）中是无法进行透视除法的，此阶段只能实现投影矩阵这一环节的运算。而透视除法将在后面交由硬件执行。

连接输入布局描述符与输入签名

根据图6.4来看，输送到渲染流水线的顶点属性与输入布局描述的定义相关联。如果我们传入的顶点数据与顶点着色器所期望的输入不相符，便会导致错误。例如，下列顶点着色器的输入签名与顶点数据就是

不匹配的：

```
//--------------
// C++应用程序代码
//--------------
struct Vertex
{
  XMFLOAT3 Pos;
  XMFLOAT4 Color;
};

D3D12_INPUT_ELEMENT_DESC desc[] =
{
  {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
  {"COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0}
};

//--------------
// 顶点着色器
//--------------
struct VertexIn
{
  float3 PosL  : POSITION;
  float4 Color : COLOR;
  float3 Normal : NORMAL;
};

struct VertexOut
{
  float4 PosH : SV_POSITION;
  float4 Color : COLOR;
};

VertexOut VS(VertexIn vin) { ... }
```

就像我们将在6.9节中看到的一样，在创建`ID3D12PipelineState`对象的时候，必须指定输入布局描述和顶点着色器，而Direct3D则会验证两者是否匹配。

事实上，顶点数据与输入签名不需要完全匹配，前提是我们一定要

向顶点着色器提供其输入签名所定义的顶点数据。这就是说，顶点数据中也可以附带一些顶点着色器根本用不到的额外数据。下面的代码就描述了这样一种匹配的情况：

```
//--------------
// C++应用程序代码
//--------------
struct Vertex
{
  XMFLOAT3 Pos;
  XMFLOAT4 Color;
  XMFLOAT3 Normal;
};

D3D12_INPUT_ELEMENT_DESC desc[] =
{
  {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
  {"COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
  { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 28,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }
};

//--------------
// 顶点着色器
//--------------
struct VertexIn
{
  float3 PosL  : POSITION;
  float4 Color : COLOR;
};

struct VertexOut
{
  float4 PosH : SV_POSITION;
  float4 Color : COLOR;
};

VertexOut VS(VertexIn vin) { ... }
```

现在，让我们再来思考这样一种情况：当顶点结构体和输入签名有

着匹配的顶点元素，唯独两者的颜色属性类型却不相同，此时会发生什么呢？

```
//--------------
// C++应用程序代码
//--------------
struct Vertex
{
  XMFLOAT3 Pos;
  XMFLOAT4 Color;
};

D3D12_INPUT_ELEMENT_DESC desc[] =
{
  {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
  {"COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0}
};

//--------------
// 顶点着色器
//--------------
struct VertexIn
{
  float3 PosL  : POSITION;
  int4 Color : COLOR;
};

struct VertexOut
{
  float4 PosH : SV_POSITION;
  float4 Color : COLOR;
};

VertexOut VS(VertexIn vin) { ... }
```

这其实也是合法的，因为Direct3D允许用户对输入寄存器（input register，全称为顶点着色器输入寄存器）中数据的类型重新加以解释。然而，VC++调试输出窗口还是会给出下面的警告：

```
D3D12 WARNING: ID3D12 Device::CreateInputLayout: The provided inputsignatu
re expects to read an element with SemanticName/Index: 'COLOR'/0and compon
ent(s) of the type 'int32'. However, the matching entry inthe Input Layout
 declaration, element[1], specifies mismatched format:'R32G32B32A32_FLOAT'
. This is not an error, since behavior is welldefined: The element format
determines what data conversion algorithmgets applied before it shows up i
n a shader register. Independently,the shader input signature defines how
the shader will interpret thedata that has been placed in its input regist
ers, with no change in thebits stored. It is valid for the application to
reinterpret data asa different type once it is in the vertex shader, so th
is warning isissued just in case reinterpretation was not intended by the
author.
```

（D3D12警告：ID3D12Device::CreateInputLayout：根据代码中提供的输入签名推断，其中的SemanticName/Index: 'COLOR'/0项希望读取'int32'类型的数据分量。但是，此类型却与输入布局声明中的对应匹配项element[1]所指定的格式'R32G32B32A32_FLOAT'不匹配。这并不是一个错误，因为Direct3D对该行为有着明确的定义：输入布局中的元素格式确定的是，在数据未进入着色器寄存器之前，应运用何种数据转换算法来确定各元素的具体格式。而着色器输入签名则定义的是，在不修改输入寄存器中所存数据的情况下，顶点着色器将如何来解释这些数据的类型。所以，在应用程序里将顶点着色器中的数据重新解释为与之不同的类型，亦是合法的。因此，该警告只用于提醒在无意间对数据类型作出不同解释的程序员[9]。）

6.5　像素着色器示例

就像在5.10.3节中所讨论的那样，为了计算出三角形中每个像素的属性，我们会在光栅化处理期间对顶点着色器（或几何着色器）输出的顶点属性进行插值。随后，再将这些插值数据传至像素着色器中作为它的输入（参见5.11节）。现假设我们的程序未使用几何着色器，图6.5展示的即为当前顶点数据所流经的路径。

像素着色器与顶点着色器有些相似：前者是针对每一个像素片段（pixel fragment）而运行的函数，后者是针对每一个顶点而运行的函数（即在每次执行时处理单个像素（顶点））。只要为像素着色器指定了输入数据，它就会为像素片段计算出一个对应的颜色。值得我们注意的是，这些输入像素着色器的像素片段有可能最终不会传入或留存在后台缓冲区中。例如，像素片段可能会在像素着色器中被裁剪掉（HLSL中内置了一个裁剪函数`clip`，可以使指定的像素片段在后续的处理流程中被忽略掉）、被另一个具有较小深度值的像素片段所遮挡或者在类似于模板缓冲区测试的后续渲染流水线测试中被丢弃。因此，在确定后台缓冲区某一像素的过程中，可能会存在多个候选的像素片段。这就是“像素片段”和“像素”意义的差别，尽管有时这两个术语可以互用，但是在一些语境下它们的意义也将变得更加分明[10]。

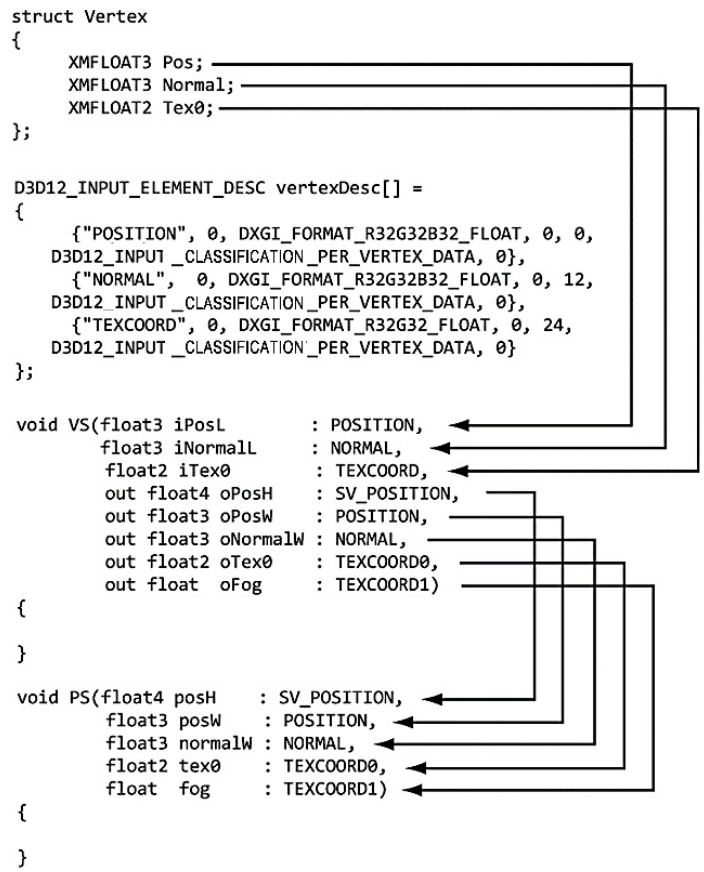


图6.5　每个顶点元素都会与D3D12_INPUT_ELEMENT_DESC数组中指定的对应语义相关联。而顶点着色器中的每个参数也各附有一个语义，用于使顶点元素与顶点着色器参数相匹配。同样地，顶点着色器的每个输出参数以及像素着色器的每个输入参数也各附有一个语义，负责把顶点着色器的输出参数映射到像素着色器的输入参数

注意

由于硬件优化的原因，某些像素片段在移送至像素着色器之前，可能已经被渲染流水线所剔除（例如提前深度剔除，early-z rejection，也有译作早期深度剔除、早期z剔除等）。这就是为什么要首先执行深度测试的原因，如果已经确定某像素片段被遮挡，那么像素着色器将不再

对它进行处理。然而，也有一些情况能够禁止提前深度剔除优化。比如说，倘若在像素着色器中有对像素深度值进行修改的操作，那么像素着色器就必须针对每个像素各执行一次，因为在像素着色器修改像素深度值以前，我们并不知道每个像素的最终深度值。

下面是一段简单的像素着色器代码，它与6.4节中给定的顶点着色器相呼应。考虑到代码的完整性，此处把顶点着色器部分也一并再次给出。

```
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorldViewProj;
};

void VS(float3 iPos : POSITION, float4 iColor : COLOR,
     out float4 oPosH : SV_POSITION,
     out float4 oColor : COLOR)
{
  // 将顶点变换到齐次裁剪空间
  oPosH = mul(float4(iPos, 1.0f), gWorldViewProj);

  // 直接把顶点颜色传递到像素着色器
  oColor = iColor;
}

float4 PS(float4 posH : SV_POSITION, float4 color : COLOR) : SV_Target
{
  return color;
}
```

在这个示例中，像素着色器只简单地返回了插值颜色数据。可以发现，像素着色器的输入与顶点着色器的输出可以准确匹配，这也是必须满足的一点。像素着色器返回一个4D颜色值，而位于此函数参数列表

后的SV_TARGET语义则表示该返回值的类型应当与渲染目标格式（render target format）相匹配（该输出值会被存于渲染目标之中）。

我们可以利用输入/输出结构体重写上述顶点着色器和像素着色器的等价实现。这与之前的表示方法有所不同，我们要将语义附加给输入/输出结构体中的成员，并通过一条用于输出结构体的返回语句代替之前的多个输出参数。

```
cbuffer cbPerObject : register(b0)
{
    float4x4 gWorldViewProj;
};

struct VertexIn
{
  float3 Pos  : POSITION;
  float4 Color : COLOR;
};

struct VertexOut
{
  float4 PosH : SV_POSITION;
  float4 Color : COLOR;
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout;

  // 把顶点变换到齐次裁剪空间
  vout.PosH = mul(float4(vin.Pos, 1.0f), gWorldViewProj);

  // 直接将顶点颜色传至像素着色器
  vout.Color = vin.Color;

  return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
  return pin.Color;
```

```
}
```

6.6　常量缓冲区

6.6.1　创建常量缓冲区

常量缓冲区（constant buffer）也是一种GPU资源（`ID3D12Resource`），其数据内容可供着色器程序所引用。就像我们在本书中将会学到的纹理等其他类型的缓冲区资源一样，它们都可以被着色器程序所引用。6.4节中的顶点着色器示例有如下代码：

```
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorldViewProj;
};
```

在这段代码中，`cbuffer`对象（常量缓冲区）的名称为`cbPerObject`，其中存储的是一个4×4矩阵`gWorldViewProj`，表示把一个点从局部空间变换到齐次裁剪空间所用到的由世界、视图和投影3种变换组合而成的矩阵。在HLSL中，可将一个4×4矩阵声明为内置的`float4x4`类型。相应地，要声明3×4矩阵或2×4矩阵，即可分别使用`float3x4`和`float2x4`这两种类型。

与顶点缓冲区和索引缓冲区不同的是，常量缓冲区通常由CPU每帧更新一次。举个例子，如果摄像机每帧都在不停地移动，那么常量缓冲区也需要在每一帧都随之以新的视图矩阵而更新。所以，我们会把常量缓冲区创建到一个上传堆而非默认堆中，这样做能使我们从CPU端更新

常量。

常量缓冲区对硬件也有特别的要求，即常量缓冲区的大小必为硬件最小分配空间（256B）的整数倍。

我们经常需要用到多个相同类型的常量缓冲区。例如，假设常量缓冲区cbPerObject内存储的是随不同物体而异的常量数据，因此，如果我们要绘制n个物体，则需要n个该类型的常量缓冲区。下列代码展示了我们是如何创建一个缓冲区资源，并利用它来存储NumElements个常量缓冲区。

```
struct ObjectConstants
{
  DirectX::XMFLOAT4X4 WorldViewProj = MathHelper::Identity4x4();
};

UINT mElementByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof
     (ObjectConstants));

ComPtr<ID3D12Resource> mUploadCBuffer;
device->CreateCommittedResource(
  &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
  D3D12_HEAP_FLAG_NONE,
  &CD3DX12_RESOURCE_DESC::Buffer(mElementByteSize * NumElements),
  D3D12_RESOURCE_STATE_GENERIC_READ,
  nullptr,
  IID_PPV_ARGS(&mUploadCBuffer));
```

我们可以认为mUploadCBuffer中存储了一个ObjectConstants类型的常量缓冲区数组（同时按256字节的整数倍来为之填充数据）。待到绘制物体的时候，只要将常量缓冲区视图（Constant Buffer View，CBV）绑定到存有物体相应常量数据的缓冲区子区域即可。由于mUploadCBuffer缓冲区存储的是一个常量缓冲区数组，因此，我们把

它称之为常量缓冲区。

工具函数d3dUtil::CalcConstantBufferByteSize会做适当的运算，使缓冲区的大小凑整为硬件最小分配空间（256B）的整数倍。

```
UINT d3dUtil::CalcConstantBufferByteSize(UINT byteSize)
{
  // 常量缓冲区的大小必须是硬件最小分配空间（通常是256B）的整数倍
  // 为此，要将其凑整为满足需求的最小的256的整数倍。我们现在通过为输入值byteSize
加上255，
  // 再屏蔽求和结果的低2字节（即计算结果中小于256的数据部分）来实现这一点
  // 例如：假设byteSize = 300
  // (300 + 255) & ~255
  // 555 & ~255
  // 0x022B & ~0x00ff
  // 0x022B & 0xff00
  // 0x0200
  // 512
  return (byteSize + 255) & ~255;
}
```

注意 Note

尽管我们已经按照上述方式在程序中分配出了256整数倍字节大小的数据空间，但是却无须为HLSL结构体中显式填充相应的常量数据，这是因为它会暗中自行完成这项工作：

```
// 隐式填充为256B
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorldViewProj;
};
```

```
// 显式填充为256B
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorldViewProj;
  float4x4 Pad0;
  float4x4 Pad1;
  float4x4 Pad2;
};
```

注意 Note

为了免去系统将常量缓冲区元素隐式凑整为256字节整数倍的这项处理环节，我们可以手动地填充所有的常量缓冲区结构体，使之皆为256字节的整数倍。

随Direct3D 12一同推出的是着色器模型（shader model，SM）[11]5.1。其中新引进了一条可用于定义常量缓冲区的HLSL语法，它的使用方法如下：

```
struct ObjectConstants
{
  float4x4 gWorldViewProj;
  uint matIndex;
};
ConstantBuffer<ObjectConstants> gObjConstants : register(b0);
```

在此段代码中，常量缓冲区的数据元素被定义在一个单独的结构体中，随后再用此结构体来创建一个常量缓冲区。这样一来，我们就可以利用下列获取数据成员的语法，在着色器里访问常量缓冲区中的各个字

段：

```
uint index = gObjConstants.matIndex;
```

6.6.2　更新常量缓冲区

由于常量缓冲区是用D3D12_HEAP_TYPE_UPLOAD这种堆类型来创建的，所以我们就能通过CPU为常量缓冲区资源更新数据。为此，我们首先要获得指向欲更新资源数据的指针，可用Map方法来做到这一点：

```
ComPtr<ID3D12Resource> mUploadBuffer;
BYTE* mMappedData = nullptr;
mUploadBuffer->Map(0, nullptr, reinterpret_cast<void**>(&mMappedData));
```

第一个参数是子资源（subresource）的索引[12]，指定了欲映射的子资源。对于缓冲区来说，它自身就是唯一的子资源，所以我们将此参数设置为0。第二个参数是一个可选项，是个指向D3D12_RANGE结构体的指针，此结构体描述了内存的映射范围，若将该参数指定为空指针，则对整个资源进行映射。第三个参数则借助双重指针，返回待映射资源数据的目标内存块。我们利用memcpy函数将数据从系统内存（system memory，也就是CPU端控制的内存）复制到常量缓冲区：

```
memcpy(mMappedData, &data, dataSizeInBytes);
```

当常量缓冲区更新完成后，我们应在释放映射内存之前对其进行Unmap（取消映射）操作[13]：

```
if(mUploadBuffer != nullptr)
  mUploadBuffer->Unmap(0, nullptr);

mMappedData = nullptr;
```

Unmap的第一个参数是子资源索引，指定了将被取消映射的子资源。若取消映射的是缓冲区，则将其置为0。第二个参数是个可选项，是一个指向D3D12_RANGE结构体的指针，用于描述取消映射的内存范围，若将它指定为空指针，则取消整个资源的映射。

6.6.3　上传缓冲区辅助函数

将上传缓冲区的相关操作简单地封装一下，使用起来会更加方便。我们在UploadBuffer.h文件中定义了下面这个类，令上传缓冲区的相关处理工作更加轻松。它替我们实现了上传缓冲区资源的构造与析构函数、处理资源的映射和取消映射操作，还提供了CopyData方法来更新缓冲区内的特定元素。在需要通过CPU修改上传缓冲区中数据的时候（例如，当观察矩阵有了变化），便可以使用CopyData。注意，此类可用于各种类型的上传缓冲区，而并非只针对常量缓冲区。当用此类管理常量缓冲区时，我们就需要通过构造函数参数isConstantBuffer来对此加以描述。另外，如果此类中存储的是常量缓冲区，那么其中的构造函数将自动填充内存，使每个常量缓冲区的大小都成为256B的整数

倍。

```
template<typename T>
class UploadBuffer
{
public:
  UploadBuffer(ID3D12Device* device, UINT elementCount,
    bool isConstantBuffer) :
    mIsConstantBuffer(isConstantBuffer)
  {
    mElementByteSize = sizeof(T);

    // 常量缓冲区的大小为256B的整数倍。这是因为硬件只能按m*256B的偏移量和n*256B的数据
    // 长度这两种规格来查看常量数据
    // typedef struct D3D12_CONSTANT_BUFFER_VIEW_DESC {
    // UINT64 OffsetInBytes; // 256的整数倍
    // UINT  SizeInBytes;    // 256的整数倍
    // } D3D12_CONSTANT_BUFFER_VIEW_DESC;
    if(isConstantBuffer)
      mElementByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof
    (T));
    ThrowIfFailed(device->CreateCommittedResource(
      &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
      D3D12_HEAP_FLAG_NONE,
      &CD3DX12_RESOURCE_DESC::Buffer(mElementByteSize*elementCount),
              D3D12_RESOURCE_STATE_GENERIC_READ,
      nullptr,
      IID_PPV_ARGS(&mUploadBuffer)));

    ThrowIfFailed(mUploadBuffer->Map(0, nullptr, reinterpret_cast<void**>
      (&mMappedData)));

    // 只要还会修改当前的资源，我们就无须取消映射
    // 但是，在资源被GPU使用期间，我们千万不可向该资源进行写操作（所以必须借助于同步技术）
  }

  UploadBuffer(const UploadBuffer& rhs) = delete;
  UploadBuffer& operator=(const UploadBuffer& rhs) = delete;
  ~UploadBuffer()
  {
    if(mUploadBuffer != nullptr)
      mUploadBuffer->Unmap(0, nullptr);

    mMappedData = nullptr;
```

```
  }

  ID3D12Resource* Resource()const
  {
    return mUploadBuffer.Get();
  }

  void CopyData(int elementIndex, const T& data)
  {
    memcpy(&mMappedData[elementIndex*mElementByteSize], &data, sizeof(T));
  }

private:
  Microsoft::WRL::ComPtr<ID3D12Resource> mUploadBuffer;
  BYTE* mMappedData = nullptr;

  UINT mElementByteSize = 0;
  bool mIsConstantBuffer = false;
};
```

一般来讲，物体的世界矩阵将随其移动/旋转/缩放而改变，观察矩阵随虚拟摄像机的移动/旋转而改变，投影矩阵随窗口大小的调整而改变。在本章的演示程序中，用户可以通过鼠标来旋转和移动摄像机，变换观察角度。因此，我们在每一帧都要用Update函数，以新的观察矩阵来更新“世界—观察—投影”3种矩阵组合而成的复合矩阵：

```
void BoxApp::OnMouseMove(WPARAM btnState, int x, int y)
{
  if((btnState & MK_LBUTTON) != 0)
  {
    // 根据鼠标的移动距离计算旋转角度，令每个像素按此角度的1/4进行旋转
    float dx = XMConvertToRadians(0.25f*static_cast<float>
    (x - mLastMousePos.x));
    float dy = XMConvertToRadians(0.25f*static_cast<float>
    (y - mLastMousePos.y));

    // 根据鼠标的输入来更新摄像机绕立方体旋转的角度
    mTheta += dx;
    mPhi += dy;

    // 限制角度mPhi的范围
    mPhi = MathHelper::Clamp(mPhi, 0.1f, MathHelper::Pi - 0.1f);
```

```
  }
  else if((btnState & MK_RBUTTON) != 0)
  {
    // 使场景中的每个像素按鼠标移动距离的0.005倍进行缩放
    float dx = 0.005f*static_cast<float>(x - mLastMousePos.x);
    float dy = 0.005f*static_cast<float>(y - mLastMousePos.y);

    // 根据鼠标的输入更新摄像机的可视范围半径
    mRadius += dx - dy;

    // 限制可视半径的范围
    mRadius = MathHelper::Clamp(mRadius, 3.0f, 15.0f);
  }

  mLastMousePos.x = x;
  mLastMousePos.y = y;
}

void BoxApp::Update(const GameTimer& gt)
{
  // 由球坐标（也有译作球面坐标）转换为笛卡儿坐标
  float x = mRadius*sinf(mPhi)*cosf(mTheta);
  float z = mRadius*sinf(mPhi)*sinf(mTheta);
  float y = mRadius*cosf(mPhi);

  // 构建观察矩阵
  XMVECTOR pos = XMVectorSet(x, y, z, 1.0f);
  XMVECTOR target = XMVectorZero();
  XMVECTOR up = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);

  XMMATRIX view = XMMatrixLookAtLH(pos, target, up);
  XMStoreFloat4x4(&mView, view);

  XMMATRIX world = XMLoadFloat4x4(&mWorld);
  XMMATRIX proj = XMLoadFloat4x4(&mProj);
  XMMATRIX worldViewProj = world*view*proj;

  // 用最新的worldViewProj 矩阵来更新常量缓冲区
  ObjectConstants objConstants;
  XMStoreFloat4x4(&objConstants.WorldViewProj,
    XMMatrixTranspose(worldViewProj));
  mObjectCB->CopyData(0, objConstants);
}
```

6.6.4 常量缓冲区描述符

在4.1.6节中，我们首次通过描述符对象将资源绑定到渲染流水线上。到目前为止，本书已经依次介绍了渲染目标、深度/模板缓冲区、顶点缓冲区以及索引缓冲区这几种资源描述符（或称视图）的使用方法，现在还需利用描述符将常量缓冲区绑定至渲染流水线上。而且常量缓冲区描述符都要存放在以D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV类型所建的描述符堆里。这种堆内可以混合存储常量缓冲区描述符、着色器资源描述符和无序访问（unordered access）描述符。为了存放这些新类型的描述符，我们需要为之创建以下类型的新式描述符堆：

```
D3D12_DESCRIPTOR_HEAP_DESC cbvHeapDesc;
cbvHeapDesc.NumDescriptors = 1;
cbvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
cbvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
cbvHeapDesc.NodeMask = 0;

ComPtr<ID3D12DescriptorHeap> mCbvHeap
md3dDevice->CreateDescriptorHeap(&cbvHeapDesc,
  IID_PPV_ARGS(&mCbvHeap));
```

这段代码与我们之前创建渲染目标和深度/模板缓冲区这两种资源描述符堆的过程很相似。然而，其中却有着一个重要的区别，那就是在创建供着色器程序访问资源的描述符时，我们要把标志Flags指定为DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE。在本章的示范程序中，我们并没有使用SRV（shader resource view，着色器资源视图）描述符或UAV（unordered access view，无序访问视图）描述符，仅是绘制了一个物体而已，因此只需创建一个存有单个CBV描述符的堆即可。

通过填写D3D12_CONSTANT_BUFFER_VIEW_DESC实例， 再调

用ID3D12Device:: CreateConstant-

BufferView方法，便可创建常量缓冲区：

```
// 绘制物体所用的常量数据
struct ObjectConstants
{
  XMFLOAT4X4 WorldViewProj = MathHelper::Identity4x4();
};

// 此常量缓冲区存储了绘制n个物体所需的常量数据
std::unique_ptr<UploadBuffer<ObjectConstants>> mObjectCB = nullptr;
mObjectCB = std::make_unique<UploadBuffer<ObjectConstants>>(
  md3dDevice.Get(), n, true);

UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(ObjectCons
tants));

// 缓冲区的起始地址(即索引为0的那个常量缓冲区的地址）
D3D12_GPU_VIRTUAL_ADDRESS cbAddress = mObjectCB->Resource()-
    >GetGPUVirtualAddress();

// 偏移到常量缓冲区中绘制第i个物体所需的常量数据
int boxCBufIndex = i;
cbAddress += boxCBufIndex*objCBByteSize;

D3D12_CONSTANT_BUFFER_VIEW_DESC cbvDesc;
cbvDesc.BufferLocation = cbAddress;
cbvDesc.SizeInBytes = d3dUtil::CalcConstantBufferByteSize(sizeof(
     ObjectConstants));

md3dDevice->CreateConstantBufferView(
  &cbvDesc,
  mCbvHeap->GetCPUDescriptorHandleForHeapStart());
```

结构体D3D12_CONSTANT_BUFFER_VIEW_DESC描述的是绑定到HLSL常量缓冲区结构体的常量缓冲区资源子集。正如前面所提到的，如果常量缓冲区存储了一个内有n个物体常量数据的常量数组，那么我们就可以通过BufferLocation和SizeInBytes参数来获取第i个物体的常量数据。考虑到硬件的需求（即硬件的最小分配空间），成

员SizeInBytes与BufferLocation必须为256B的整数倍。例如，若将上述两个成员的值都指定为64，那么我们将看到下列调试错误：

```
D3D12 ERROR: ID3D12Device::CreateConstantBufferView: SizeInBytes of 64 is
invalid. Device requires SizeInBytes be a multiple of 256.
D3D12 ERROR: ID3D12Device:: CreateConstantBufferView: BufferLocation of 64
 is invalid. Device requires BufferLocation be a multiple of 256.
```

（D3D12错误：ID3D12Device::CreateConstantBufferView: 将SizeInBytes的值设置为64是无效的。设备要求SizeInBytes的值为256的整数倍。

D3D12错误:ID3D12Device:: CreateConstantBufferView: 将BufferLocation的值设置为64是无效的。设备要求BufferLocation的值为256的整数倍。）

6.6.5　根签名和描述符表

通常来讲，在绘制调用开始执行之前，我们应将不同的着色器程序所需的各种类型的资源绑定到渲染流水线上。实际上，不同类型的资源会被绑定到特定的寄存器槽（register slot）上，以供着色器程序访问。比如说，前文代码中的顶点着色器和像素着色器需要的是一个绑定到寄存器b0的常量缓冲区。在本书的后续内容中，我们会用到这两种着色器更高级的配置方法，以使多个常量缓冲区、纹理（texture）和采样器（sampler）都能与各自的寄存器槽相绑定[14]：

```
// 将纹理资源绑定到纹理寄存器槽0
Texture2D  gDiffuseMap : register(t0);
```

```
// 把下列采样器资源依次绑定到采样器寄存器槽0~5
SamplerState gsamPointWrap        : register(s0);
SamplerState gsamPointClamp       : register(s1);
SamplerState gsamLinearWrap       : register(s2);
SamplerState gsamLinearClamp      : register(s3);
SamplerState gsamAnisotropicWrap  : register(s4);
SamplerState gsamAnisotropicClamp : register(s5);

// 将常量缓冲区资源（cbuffer）绑定到常量缓冲区寄存器槽0
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorld;
  float4x4 gTexTransform;
};

// 绘制过程中所用的杂项常量数据
cbuffer cbPass : register(b1)
{
  float4x4 gView;
  float4x4 gProj;
  [...] // 为篇幅而省略的其他字段
};

// 绘制每种材质所需的各种不同的常量数据
cbuffer cbMaterial : register(b2)
{
  float4  gDiffuseAlbedo;
  float3  gFresnelR0;
  float  gRoughness;
  float4x4 gMatTransform;
};
```

根签名（root signature）定义的是：在执行绘制命令之前，那些应用程序将绑定到渲染流水线上的资源，它们会被映射到着色器的对应输入寄存器。根签名一定要与使用它的着色器相兼容（即在绘制开始之前，根签名一定要为着色器提供其执行期间需要绑定到渲染流水线的所有资源），在创建流水线状态对象（pipeline state object）时会对此进行验证（参见6.9节）。不同的绘制调用可能会用到一组不同的着色器程序，这也就意味着要用到不同的根签名。

注意 Note

如果我们把着色器程序当作一个函数，而将输入资源看作着色器的函数参数，那么根签名则定义了函数签名（其实这就是“根签名”一词的由来）。通过绑定不同的资源作为参数，着色器的输出也将有所差别。例如，顶点着色器的输出取决于实际向它输入的顶点数据以及为它绑定的具体资源。

在Direct3D中，根签名由`ID3D12RootSignature`接口来表示，并以一组描述绘制调用过程中着色器所需资源的根参数（root parameter）定义而成。根参数可以是根常量（root constant）、根描述符（root descriptor）或者描述符表（descriptor table）。我们在本章中仅使用描述符表，其他根参数均在第7章中进行讨论。描述符表指定的是描述符堆中存有描述符的一块连续区域。

下面的代码创建了一个根签名，它的根参数为一个描述符表，其大小足以容下一个CBV（常量缓冲区视图，constant buffer view）。

```
// 根参数可以是描述符表、根描述符或根常量
CD3DX12_ROOT_PARAMETER slotRootParameter[1];

// 创建一个只存有一个CBV的描述符表
CD3DX12_DESCRIPTOR_RANGE cbvTable;
cbvTable.Init(
  D3D12_DESCRIPTOR_RANGE_TYPE_CBV,
  1, // 表中的描述符数量
```

```
  0);// 将这段描述符区域绑定至此基准着色器寄存器（base shader register）

slotRootParameter[0].InitAsDescriptorTable(
  1,            // 描述符区域的数量
  &cbvTable); // 指向描述符区域数组的指针

// 根签名由一组根参数构成
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(1, slotRootParameter, 0, nullptr,
  D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

// 创建仅含一个槽位（该槽位指向一个仅由单个常量缓冲区组成的描述符区域）的根签名
ComPtr serializedRootSig = nullptr;
ComPtr errorBlob = nullptr;
HRESULT hr = D3D12SerializeRootSignature(&rootSigDesc,
  D3D_ROOT_SIGNATURE_VERSION_1,
  serializedRootSig.GetAddressOf(),
  errorBlob.GetAddressOf());[15]
ThrowIfFailed(md3dDevice->CreateRootSignature(
  0,
  serializedRootSig->GetBufferPointer(),
  serializedRootSig->GetBufferSize(),
  IID_PPV_ARGS(&mRootSignature)));
```

我们将在第7章对CD3DX12_ROOT_PARAMETER 和CD3DX12_DESCRIPTOR_RANGE这两种辅助结构进行更加细致的解读，现在只需理解下述代码即可：

```
CD3DX12_ROOT_PARAMETER slotRootParameter[1];

CD3DX12_DESCRIPTOR_RANGE cbvTable;
cbvTable.Init(
  D3D12_DESCRIPTOR_RANGE_TYPE_CBV, // 描述符表的类型
  1, // 表中描述符的数量
  0);// 将这段描述符区域绑定至此基址着色器寄存器
slotRootParameter[0].InitAsDescriptorTable(
  1,            // 描述符区域的数量
  &cbvTable); // 指向描述符区域数组的指针
```

这段代码创建了一个根参数，目的是将含有一个CBV的描述符表绑定到常量缓冲区寄存器0，即HLSL代码中的register(b0)。

注意 Note

本章中所展示的根签名示例十分简单。读者在此书的大部分范例中会见到根签名的身影，而且根签名的复杂度也将按程序的需求而逐渐提升。

根签名只定义了应用程序要绑定到渲染流水线的资源，却没有真正地执行任何资源绑定操作。只要率先通过命令列表（command list）设置好根签名，我们就能用`ID3D12GraphicsCommandList::SetGraphicsRootDescriptorTab`方法令描述符表与渲染流水线相绑定。

```
void ID3D12GraphicsCommandList::SetGraphicsRootDescriptorTable(
  UINT RootParameterIndex,
  D3D12_GPU_DESCRIPTOR_HANDLE BaseDescriptor);
```

1. `RootParameterIndex`：将根参数按此索引（即欲绑定到的寄存器槽号）进行设置。

2. `BaseDescriptor`：此参数指定的是将要向着色器绑定的描述符表中第一个描述符位于描述符堆中的句柄。比如说，如果根签名指明当前描述符表中共有 5 个描述符，则堆中的`BaseDescriptor`及其后面的4个描述符将被设置到此描述符表中。

下列代码先将根签名和CBV堆设置到命令列表上，并随后再通过设置描述符表来指定我们希望绑定到渲染流水线的资源：

```
mCommandList->SetGraphicsRootSignature(mRootSignature.Get());
ID3D12DescriptorHeap* descriptorHeaps[] = { mCbvHeap.Get() };
mCommandList->SetDescriptorHeaps(_countof(descriptorHeaps),
    descriptorHeaps);

// 偏移到此次绘制调用所需的CBV处
CD3DX12_GPU_DESCRIPTOR_HANDLE cbv(mCbvHeap
    ->GetGPUDescriptorHandleForHeapStart());
cbv.Offset(cbvIndex, mCbvSrvUavDescriptorSize);

mCommandList->SetGraphicsRootDescriptorTable(0, cbv);
```

注意 Note

出于性能的原因，我们应当使根签名的规模尽可能地小。除此之外，还要试着尽量减少每帧渲染过程中根签名的修改次数。

注意 Note

每当在（图形）绘制调用或（计算）调度（dispatch，也有译作分派）调用（此“调度调用”指调度计算着色器进行GPU通用计算）之间有根签名的内容（即描述符表、根常量以及根描述符）发生改变时，D3D12的驱动程序便会将与应用程序相绑定的根签名内容自动更新为最

新的数据。因此，在每次绘制/调度调用时都会产生一整套独立的根签名状态[16]。

注意 Note

如果更改了根签名，则会失去现存的所有绑定关系。也就是说，在修改了根签名后，我们需要按新的根签名定义重新将所有的对应资源绑定到渲染流水线上。

6.7 编译着色器

在Direct3D中，着色器程序必须先被编译为一种可移植的字节码。接下来，图形驱动程序将获取这些字节码，并将其重新编译为针对当前系统GPU所优化的本地指令[ATI1]。我们可以在运行期间用下列函数对着色器进行编译。

```
HRESULT D3DCompileFromFile(
  LPCWSTR pFileName,
  const D3D_SHADER_MACRO *pDefines,
  ID3DInclude *pInclude,
  LPCSTR pEntrypoint,
  LPCSTR pTarget,
  UINT Flags1,
  UINT Flags2,
  ID3DBlob **ppCode,
  ID3DBlob **ppErrorMsgs);
```

1．`pFileName`：我们希望编译的以.hlsl作为扩展名的HLSL源代码文件。

2．`pDefines`：在本书中，我们并不使用这个高级选项，因此总是将它指定为空指针。关于此参数的详细信息可参见SDK文档。

3．`pInclude`：在本书中，我们并不使用这个高级选项，因而总是将它指定为空指针。关于此参数的详细信息可详见SDK文档。

4．`pEntrypoint`：着色器的入口点函数名。一个.hlsl文件可能存有多个着色器程序（例如，一个顶点着色器和一个像素着色器），所以我们需要为待编译的着色器指定入口点。

5. pTarget：指定所用着色器类型和版本的字符串。在本书中，我们采用的着色器模型版本是5.0和5.1[17]。

a）vs_5_0与vs_5_1：表示版本分别为5.0和5.1的顶点着色器（vertex shader）。

b）hs_5_0与hs_5_1：表示版本分别为5.0和5.1的外壳着色器（hull shader）。

c）ds_5_0与ds_5_1：表示版本分别为5.0和5.1的域着色器（domain shader）。

d）gs_5_0与gs_5_1：表示版本分别为5.0和5.1的几何着色器（geometry shader）。

e）ps_5_0与ps_5_1：表示版本分别为5.0和5.1的像素着色器（pixel shader）。

f）cs_5_0与cs_5_1：表示版本分别为5.0和5.1的计算着色器（compute shader）。

6. Flags1：指示对着色器代码应当如何编译的标志。在SDK文档里，这些标志列出得不少，但是此书中我们仅用两种。

a）**D3DCOMPILE_DEBUG**：用调试模式来编译着色器。

b）**D3DCOMPILE_SKIP_OPTIMIZATION**：指示编译器跳过优化阶段

（对调试很有用处）。

7. Flags2：我们不会用到处理效果文件的高级编译选项，关于它的信息请参见SDK文档。

8. ppCode：返回一个指向ID3DBlob数据结构的指针，它存储着编译好的着色器对象字节码。

9. ppErrorMsgs：返回一个指向ID3DBlob数据结构的指针。如果在编译过程中发生了错误，它便会储存报错的字符串。

ID3DBlob类型描述的其实就是一段普通的内存块，这是该接口的两个方法：

a）LPVOID GetBufferPointer：返回指向ID3DBlob对象中数据的void*类型的指针。由此可见，在使用此数据之前务必先要将它转换为适当的类型（参考下面的示例）。

b）SIZE_T GetBufferSize：返回缓冲区的字节大小（即该对象中的数据大小）。

为了能够输出错误信息，我们在d3dUtil.h/.cpp文件中实现了下列辅助函数在运行时编译着色器：

```
ComPtr<ID3DBlob> d3dUtil::CompileShader(
    const std::wstring& filename,
    const D3D_SHADER_MACRO* defines,
    const std::string& entrypoint,
    const std::string& target)
{
```

```
  // 若处于调试模式,则使用调试标志
  UINT compileFlags = 0;
#if defined(DEBUG) || defined(_DEBUG)
  compileFlags = D3DCOMPILE_DEBUG | D3DCOMPILE_SKIP_OPTIMIZATION;
#endif

  HRESULT hr = S_OK;

  ComPtr<ID3DBlob> byteCode = nullptr;
  ComPtr<ID3DBlob> errors;
  hr = D3DCompileFromFile(filename.c_str(), defines,
     D3D_COMPILE_STANDARD_FILE_INCLUDE,
    entrypoint.c_str(), target.c_str(), compileFlags, 0, &byteCode,
    &errors);

  // 将错误信息输出到调试窗口
  if(errors != nullptr)
    OutputDebugStringA((char*)errors->GetBufferPointer());

  ThrowIfFailed(hr);

  return byteCode;
}
```

以下是一个调用此函数的示例：

```
ComPtr<ID3DBlob> mvsByteCode = nullptr;
ComPtr<ID3DBlob> mpsByteCode = nullptr;
mvsByteCode = d3dUtil::CompileShader(L"Shaders\\color.hlsl",
  nullptr, "VS", "vs_5_0");
mpsByteCode = d3dUtil::CompileShader(L"Shaders\\color.hlsl",
  nullptr, "PS", "ps_5_0");
```

HLSL的错误和警告消息将通过ppErrorMsgs参数返回。比方说，如果不小心把mul函数拼写错误，那么我们便会从调试窗口得到类似于下列的错误输出：

```
Shaders\color.hlsl(29,14-55): error X3004: undeclared identifier 'mu'
（Shaders\color.hlsl(29,14-55): 错误 X3004: 未声明的标识符 'mu'）
```

仅对着色器进行编译并不会使它与渲染流水线相绑定以供其使用，我们将在6.9节中介绍相关的具体做法。

6.7.1 离线编译

我们不仅可以在运行期间编译着色器，还能够以单独的步骤（例如，将其作为构建整个工程过程中的一个独立环节，或是将其视为资源内容流水线（asset content pipeline）流程的一部分）离线地（offline）编译着色器。这样做有原因若干：

1．对于复杂的着色器来说，其编译过程可能耗时较长。因此，借助离线编译即可缩短应用程序的加载时间。

2．以便在早于运行时的构建处理期间提前发现编译错误。

3．对于Windows 8应用商店中的应用而言，必须采用离线编译这种方式。

我们通常用.cso (即compiled shader object，已编译的着色器对象)作为已编译着色器的扩展名。

为了以离线的方式编译着色器，我们将使用DirectX自带的FXC命令行编译工具。为了将color.hlsl文件中分别以VS和PS作为入口点的顶点着色器和像素着色器编译为调试版本的字节码，我们可以输入以下命令：

```
fxc "color.hlsl" /Od /Zi /T vs_5_0 /E "VS" /Fo "color_vs.cso" /Fc "color_v
s.asm"
fxc "color.hlsl" /Od /Zi /T ps_5_0 /E "PS" /Fo "color_ps.cso" /Fc "color_p
```

```
s.asm"
```

为了将color.hlsl文件中分别以VS和PS作为入口点的顶点着色器和像素着色器编译为发行版本的字节码，则可以输入以下命令：

```
fxc "color.hlsl" /T vs_5_0 /E "VS" /Fo "color_vs.cso" /Fc "color_vs.asm"
fxc "color.hlsl" /T ps_5_0 /E "PS" /Fo "color_ps.cso" /Fc "color_ps.asm"
```

参数	描述
/Od	禁用优化（对于调试十分有用）
/Zi	开启调试信息
/T <string>	着色器类型和着色器模型的版本
/E <string>	着色器入口点
/Fo <string>	经过编译的着色器对象字节码
/Fc <string>	输出一个着色器的汇编文件清单（对于调试、检验指令数量、查阅生成的代码细节都是很有帮助的）

如果试图编译一个有语法错误的着色器，则FXC会将错误/警告消

息输出到命令窗口。譬如，若是在color.hlsl效果文件中拼错了一个变量的名字：

```
// 应当为gWorldViewProj，而非worldViewProj
vout.PosH = mul(float4(vin.Pos, 1.0f), worldViewProj);
```

那么，我们会因为这一个失误而从调试输出窗口收到许多错误信息（关键在于改正最上面的错误）：

color.hlsl(29,42-54): error X3004: undeclared identifier

'worldViewProj'

color.hlsl(29,14-55): error X3013: 'mul': no matching 2 parameter

intrinsic function

color.hlsl(29,14-55): error X3013: Possible intrinsic functions are:

color.hlsl(29,14-55): error X3013: mul(float|half...

（color.hlsl(29,42-54)：错误X3004:未声明的标识符'worldViewProj'

color.hlsl(29,14-55)：错误X3013:'mul'： 没有比对到该内置函数所需的两个参数

color.hlsl(29,14-55)：错误X3013:可能的内置函数是：

color.hlsl(29,14-55)：错误X3013: mul(float|half...blabla...)

可见，在编译期间及时获取错误信息要比在运行时才获得错误消息要便捷得多。

既然已经按离线的方式把顶点着色器和像素着色器编译到.cso文件里，也就不需要在运行时对其进行编译（即，无须再调用D3DCompileFromFile方法）。但是，我们仍要将.cso文件中已编译好的着色器对象字节码加载到应用程序中，这可以由C++的标准文件输入机制来加以实现，如：

```
ComPtr<ID3DBlob> d3dUtil::LoadBinary(const std::wstring& filename)
{
  std::ifstream fin(filename, std::ios::binary);

  fin.seekg(0, std::ios_base::end);
  std::ifstream::pos_type size = (int)fin.tellg();
  fin.seekg(0, std::ios_base::beg);

  ComPtr<ID3DBlob> blob;
  ThrowIfFailed(D3DCreateBlob(size, blob.GetAddressOf()));

  fin.read((char*)blob->GetBufferPointer(), size);
  fin.close();

  return blob;
}
...
ComPtr<ID3DBlob> mvsByteCode = d3dUtil::LoadBinary(L"Shaders\\color_vs.cso
");
ComPtr<ID3DBlob> mpsByteCode = d3dUtil::LoadBinary(L"Shaders\\color_ps.cso
");
```

6.7.2 生成着色器汇编代码

FXC程序根据可选参数/Fc 来生成可移植的着色器汇编代码。通过查阅着色器的汇编代码，既可核对着色器的指令数量，也能了解生成的代码细节——这是为了验证编译器所生成的代码与我们预想的是否一致。例如，如果我们在HLSL代码中写了一个条件语句，那么可能会认为汇编代码中将存在一条与之对应的分支指令。在可编程GPU发展的初期阶段中，在着色器里使用分支指令的代价是比较高昂的。因此，编译器时常会通过对两个分支展开求值，再对求值结果进行插值来整理条件语句，以避免采用分支指令并计算出正确的结果。例如，下列两组代码是等价的：

条件语句	整理后
`float x = 0;` `// s == 1 (true) or s == 0 (false) if( s )` `  x = sqrt(y);` `else` `  x = 2*y;`	`float a = 2*y;` `float b = sqrt(y); float x = a + s*(b-a);` `// s == 1: x = a + b – a = b = sqrt(y)` `// s == 0: x = a + 0*(b-a) = a = 2*y`

因此，若采用这种展开整理方法，我们将得到没有任何分支语句而效果却又与整理前相同的代码。但是，在不查阅着色器汇编代码的情况下，我们无法知道此展开过程是否发生，甚至不能验证生成的分支指令

是否正确。有时，查看着色器汇编代码的目的是为了弄清它到底做了什么。下面就是一个由color.hlsl文件中顶点着色器生成的汇编代码示例：

```
//
// 生成自微软(R) HLSL着色器编译器 6.4.9844.0
//
//
// 缓冲区定义
//
// cbuffer cbPerObject
// {
//
//  float4x4 gWorldViewProj;      // 偏移量:  0 大小:  64
//
// }
//
//
// 资源绑定
//
// 名称           类型      格式     维度    槽   元素
// ------------ -------- ------ ----- --- ------- ----------
// cbPerObject  cbuffer  NA     NA    0    1
//
//
//
// 输入签名
//
// 名称            索引          掩码    寄存器   系统值     格式       使用情况
// --------      ---------- ----- ------ -------- -------- ---------
// POSITION      0            xyz    0      NONE     float    xyz
// COLOR         0            xyzw  1      NONE     float    xyzw
//
//
// 输出签名
//
// 名称           索引            掩码    寄存器   系统值     格式       使用情况
// --------     -----------   ----- ------ -------- -------- -------
// SV_POSITION  0              xyzw  0      POS      float    xyzw
// COLOR        0              xyzw  1      NONE     float    xyzw
//
vs_5_0
dcl_globalFlags refactoringAllowed | skipOptimization
dcl_constantbuffer cb0[4], immediateIndexed
dcl_input v0.xyz
dcl_input v1.xyzw
```

```
dcl_output_siv o0.xyzw, position
dcl_output o1.xyzw
dcl_temps 2
//
// 初始化变量关系
//  v0.x <- vin.PosL.x; v0.y <- vin.PosL.y; v0.z <- vin.PosL.z;
//  v1.x <- vin.Color.x; v1.y <- vin.Color.y; v1.z <- vin.Color.z; v1.w <-
 vin.Color.w;
//  o1.x <- <VS return value>.Color.x;
//  o1.y <- <VS return value>.Color.y;
//  o1.z <- <VS return value>.Color.z;
//  o1.w <- <VS return value>.Color.w;
//  o0.x <- <VS return value>.PosH.x;
//  o0.y <- <VS return value>.PosH.y;
//  o0.z <- <VS return value>.PosH.z;
//  o0.w <- <VS return value>.PosH.w
//
#第29行"color.hlsl"
mov r0.xyz, v0.xyzx
mov r0.w, l(1.000000)
dp4 r1.x, r0.xyzw, cb0[0].xyzw // r1.x <- vout.PosH.x
dp4 r1.y, r0.xyzw, cb0[1].xyzw // r1.y <- vout.PosH.y
dp4 r1.z, r0.xyzw, cb0[2].xyzw // r1.z <- vout.PosH.z
dp4 r1.w, r0.xyzw, cb0[3].xyzw // r1.w <- vout.PosH.w

#第32行
mov r0.xyzw, v1.xyzw // r0.x <- vout.Color.x; r0.y <- vout.Color.y;
           // r0.z <- vout.Color.z; r0.w <- vout.Color.w
mov o0.xyzw, r1.xyzw
mov o1.xyzw, r0.xyzw
ret
// 大约使用了10个指令槽
```

6.7.3 利用Visual Studio离线编译着色器

Visual Studio 2015集成了一些对着色器程序进行编译工作的支持。我们可以向工程内添加.hlsl文件，而Visual Studio (VS)会识别它们并提供编译的选项（见图6.6）。这些在UI中配置的选项就是FXC程序的参数。在向VS工程中添加HLSL文件后，它将成为构建流程的一部分，而

着色器也将会被FXC程序所编译。

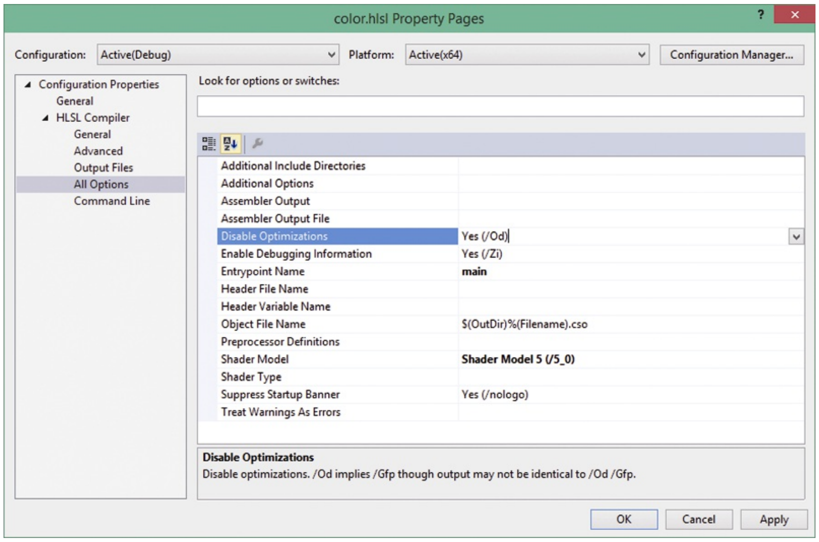


图6.6　为项目添加一个自定义的构建工具

但是，使用VS集成的HLSL工具却有一个缺点，即它只允许每个文件中仅有一个着色器程序。因此，这条限制将令顶点着色器和像素着色器不能共存于一个文件里。此外，我们有时希望以不同的预处理指令（preprocessor directives）编译同一个着色器程序，从而获取同一着色器的不同编译结果。同样地，如果使用集成的VS工具就不可能做到这一点，因为每输入一个.hlsl文件则只能输出一个.cso文件。

6.8 光栅器状态

当今渲染流水线中的大多阶段都是可编程的，但是有些特定环节却只能接受配置。例如，用于配置渲染流水线中光栅化阶段的光栅器状态（rasterizer state）组由结构体D3D12_RASTERIZER_DESC来表示：

```
typedef struct D3D12_RASTERIZER_DESC {
  D3D12_FILL_MODE FillMode;      // 默认值为: D3D12_FILL_MODE_SOLID
  D3D12_CULL_MODE CullMode;      // 默认值为: D3D12_CULL_MODE_BACK
  BOOL FrontCounterClockwise;    // 默认值为: false
  INT DepthBias;                 // 默认值为: 0
  FLOAT DepthBiasClamp;          // 默认值为: 0.0f
  FLOAT SlopeScaledDepthBias;    // 默认值为: 0.0f
  BOOL DepthClipEnable;          // 默认值为: true
  BOOL MultisampleEnable;        // 默认值为: false
  BOOL AntialiasedLineEnable;    // 默认值为: false
  UINT ForcedSampleCount;        // 默认值为: 0

  // 默认值为: D3D12_CONSERVATIVE_RASTERIZATION_MODE_OFF
  D3D12_CONSERVATIVE_RASTERIZATION_MODE ConservativeRaster;
} D3D12_RASTERIZER_DESC;
```

其中大部分对于我们而言是相对高级或不常使用的成员，我们可以从SDK文档中查阅到它们的相关描述。下面仅对其中关键的3个成员进行讲解。

1. FillMode：将此参数设置为D3D12_FILL_MODE_WIREFRAME是采用线框模式进行渲染，而设置为D3D12_FILL_MODE_SOLID则是使用实体模式进行渲染。默认设置为实体渲染模式。

2. CullMode：指定D3D12_CULL_MODE_NONE是禁用剔除操作，D3D12_CULL_MODE_BACK是剔除背面朝向的三角形，

而D3D12_CULL_MODE_FRONT是剔除正面朝向的三角形。默认配置为剔除背面朝向的三角形。

3. FrontCounterClockwise：如果指定为false，则根据摄像机的观察视角，将顶点顺序为顺时针方向的三角形看作正面朝向，而把逆时针绕序的三角形当作背面朝向。相反，如果指定为true，则根据摄像机的观察视角，将顶点顺序为逆时针方向的三角形看作正面朝向，而把顺时针绕序的三角形当作背面朝向。此参数默认值为false。

下列代码展示了如何创建一个开启线框模式，且禁用背面剔除的光栅化状态：

```
CD3DX12_RASTERIZER_DESC rsDesc(D3D12_DEFAULT);
rsDesc.FillMode = D3D12_FILL_MODE_WIREFRAME;
rsDesc.CullMode = D3D12_CULL_MODE_NONE;
```

CD3DX12_RASTERIZER_DESC是在扩展自D3D12_RASTERIZER_DESC结构体的基础上，又添加了一些辅助构造函数的工具类。其中有一个以接收CD3DX12_DEFAULT作为参数来创建光栅化状态对象的构造函数，其实CD3DX12_DEFAULT只是一个哑类型（dummy），而此函数的作用是将光栅化状态中需要被初始化的成员重载为默认值。CD3DX12_DEFAULT[18]和D3D12_DEFAULT的定义如下：

```
struct CD3DX12_DEFAULT {};
extern const DECLSPEC_SELECTANY CD3DX12_DEFAULT D3D12_DEFAULT;
```

另外，D3D12_DEFAULT（CD3DX12_DEFAULT）还被广泛地运用于Direct3D的其他几种工具类中[19]。

6.9　流水线状态对象

到目前为止，我们已经展示过编写输入布局描述、创建顶点着色器和像素着色器，以及配置光栅器状态组这3个步骤，还未曾讲解如何将这些对象绑定到图形流水线上，用以实际绘制图形。大多数控制图形流水线状态的对象被统称为流水线状态对象（Pipeline State Object，PSO），用ID3D12PipelineState接口来表示。要创建PSO，我们首先要填写一份描述其细节的D3D12_GRAPHICS_PIPELINE_STATE_DESC结构体实例。

```
typedef struct D3D12_GRAPHICS_PIPELINE_STATE_DESC
{
  ID3D12RootSignature *pRootSignature;
  D3D12_SHADER_BYTECODE VS;
  D3D12_SHADER_BYTECODE PS;
  D3D12_SHADER_BYTECODE DS;
  D3D12_SHADER_BYTECODE HS;
  D3D12_SHADER_BYTECODE GS;
  D3D12_STREAM_OUTPUT_DESC StreamOutput;
  D3D12_BLEND_DESC BlendState;
  UINT SampleMask;
  D3D12_RASTERIZER_DESC RasterizerState;
  D3D12_DEPTH_STENCIL_DESC DepthStencilState;
  D3D12_INPUT_LAYOUT_DESC InputLayout;
  D3D12_PRIMITIVE_TOPOLOGY_TYPE PrimitiveTopologyType;
  UINT NumRenderTargets;
  DXGI_FORMAT RTVFormats[8];
  DXGI_FORMAT DSVFormat;
  DXGI_SAMPLE_DESC SampleDesc;
} D3D12_GRAPHICS_PIPELINE_STATE_DESC;[20]
```

1. pRootSignature：指向一个与此PSO相绑定的根签名的指针。该根签名一定要与此PSO指定的着色器相兼容。

2. VS：待绑定的顶点着色器。此成员由结构体D3D12_SHADER_BYTECODE表示，这个结构体存有指向已编译好的字节码数据的指针，以及该字节码数据所占的字节大小。

```
typedef struct D3D12_SHADER_BYTECODE {
 const void *pShaderBytecode;
 SIZE_T   BytecodeLength;
} D3D12_SHADER_BYTECODE;
```

3. PS：待绑定的像素着色器。

4. DS：待绑定的域着色器（我们将在后续章节中讲解此类型的着色器）。

5. HS：待绑定的外壳着色器（我们将在后续章节中讲解此类型的着色器）。

6. GS：待绑定的几何着色器（我们将在后续章节中讲解此类型的着色器）。

7. StreamOutput：用于实现一种称作流输出（stream-out）的高级技术。目前我们仅将此字段清零。

8. BlendState：指定混合（blending）操作所用的混合状态。我们将在后续章节中讨论此状态组，目前仅将此成员指定为默认的CD3DX12_BLEND_DESC(D3D12_DEFAULT)。

9. SampleMask：多重采样最多可采集32个样本。借此参数的32位

整数值，即可设置每个采样点的采集情况（采集或禁止采集）。例如，若禁用了第5位（将第5位设置为0），则将不会对第5个样本进行采样。当然，要禁止采集第5 个样本的前提是，所用的多重采样至少要有5个样本。假如一个应用程序仅使用了单采样（single sampling），那么只能针对该参数的第1位进行配置。一般来说，使用的都是默认值0xffffffff，即表示对所有的采样点都进行采样。

10. RasterizerState：指定用来配置光栅器的光栅化状态。

11. DepthStencilState：指定用于配置深度/模板测试的深度/模板状态。我们将在后续章节中对此状态进行讨论，目前只把它设为默认的CD3DX12_DEPTH_STENCIL_DESC(D3D12_DEFAULT)。

12. InputLayout：输入布局描述，此结构体中有两个成员：一个由D3D12_INPUT_ELEMENT_DESC元素构成的数组，以及一个表示此数组中元素数量的无符号整数。

```
typedef struct D3D12_INPUT_LAYOUT_DESC
{
  const D3D12_INPUT_ELEMENT_DESC *pInputElementDescs;
  UINT NumElements;
} D3D12_INPUT_LAYOUT_DESC;
```

13. PrimitiveTopologyType：指定图元的拓扑类型。

```
typedef enum D3D12_PRIMITIVE_TOPOLOGY_TYPE {
 D3D12_PRIMITIVE_TOPOLOGY_TYPE_UNDEFINED = 0,
 D3D12_PRIMITIVE_TOPOLOGY_TYPE_POINT   = 1,
 D3D12_PRIMITIVE_TOPOLOGY_TYPE_LINE    = 2,
 D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE  = 3,
 D3D12_PRIMITIVE_TOPOLOGY_TYPE_PATCH   = 4
} D3D12_PRIMITIVE_TOPOLOGY_TYPE;
```

14. NumRenderTargets：同时所用的渲染目标数量（即RTVFormats数组中渲染目标格式的数量）。

15. RTVFormats：渲染目标的格式。利用该数组实现向多渲染目标同时进行写操作。使用此PSO的渲染目标的格式设定应当与此参数相匹配。

16. DSVFormat：深度/模板缓冲区的格式。使用此PSO的深度/模板缓冲区的格式设定应当与此参数相匹配。

17. SampleDesc：描述多重采样对每个像素采样的数量及其质量级别。此参数应与渲染目标的对应设置相匹配。

在D3D12_GRAPHICS_PIPELINE_STATE_DESC实例填写完毕后，我们即可用ID3D12Device::CreateGraphicsPipelineState方法来创建ID3D12PipelineState对象。

```
ComPtr mRootSignature;
std::vector mInputLayout;
ComPtr mvsByteCode;
ComPtr mpsByteCode;
...
D3D12_GRAPHICS_PIPELINE_STATE_DESC psoDesc;
ZeroMemory(&psoDesc, sizeof(D3D12_GRAPHICS_PIPELINE_STATE_DESC));
psoDesc.InputLayout = { mInputLayout.data(), (UINT)mInputLayout.size() };
psoDesc.pRootSignature = mRootSignature.Get();
psoDesc.VS =
{
  reinterpret_cast(mvsByteCode->GetBufferPointer()),
  mvsByteCode->GetBufferSize()
};
psoDesc.PS =
{
```

```
  reinterpret_cast(mpsByteCode->GetBufferPointer()),
  mpsByteCode->GetBufferSize()
};
psoDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);
psoDesc.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT);
psoDesc.DepthStencilState = CD3DX12_DEPTH_STENCIL_DESC(D3D12_DEFAULT);
psoDesc.SampleMask = UINT_MAX;
psoDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE;
psoDesc.NumRenderTargets = 1;
psoDesc.RTVFormats[0] = mBackBufferFormat;
psoDesc.SampleDesc.Count = m4xMsaaState ? 4 : 1;
psoDesc.SampleDesc.Quality = m4xMsaaState ? (m4xMsaaQuality - 1) : 0;
psoDesc.DSVFormat = mDepthStencilFormat;

ComPtr mPSO;
md3dDevice->CreateGraphicsPipelineState(&psoDesc, IID_PPV_ARGS(&mPSO));
```
[21]

ID3D12PipelineState对象集合了大量的流水线状态信息。为了保证性能，我们将所有这些对象都集总在一起，一并送至渲染流水线。通过这样的一个集合，Direct3D便可以确定所有的状态是否彼此兼容，而驱动程序则能够据此而提前生成硬件本地指令及其状态。在Direct3D 11的状态模型中，这些渲染状态片段都是要分开配置的。然而这些状态实际都有一定的联系，以致如果其中的一个状态发生改变，那么驱动程序可能就要为了另一个相关的独立状态而对硬件重新进行编程。由于一些状态在配置流水线时需要改变，因而硬件状态也就可能被频繁地改写。为了避免这些冗余的操作，驱动程序往往会推迟针对硬件状态的编程动作，直到明确整条流水线的状态发起绘制调用后，才正式生成对应的本地指令与状态。但是，这种延迟操作需要驱动在运行时进行额外的记录工作，即追踪状态的变化，而后才能在运行时生成改写硬件状态的本地代码。在Direct3D 12的新模型中，驱动程序可以在初始化期间生成对流水线状态编程的全部代码，这便是我们将大多数的流水线状态指定为一个集合所带来的好处。

注意 Note

由于PSO的验证和创建操作过于耗时，所以应在初始化期间就生成PSO。除非有特别的需求，例如，在运行时创建PSO伊始就要当即对它进行第一次引用的这种情况。随后，我们就可将它存于如散列表（哈希表）这样的集合里，以便在后续使用时快速获取。

并非所有的渲染状态都封装于PSO内，如视口（viewport）和裁剪矩形（scissor rectangle）等属性就独立于PSO。由于将这些状态的设置与其他的流水线状态分隔开来会更有效，所以把它们强行集中在PSO内也并不会为之增添任何优势。

Direct3D实质上就是一种状态机（state machine），里面的事物会保持它们各自的状态，直到我们将其改变。如果我们以不同的PSO去绘制不同物体，则需要像下面那样来组织代码：

```
// 重置命令列表并指定初始PSO
mCommandList->Reset(mDirectCmdListAlloc.Get(), mPSO1.Get());
/* ……使用PSO 1绘制物体…… */

// 改变PSO
mCommandList->SetPipelineState(mPSO2.Get());
/* ……使用PSO 2绘制物体…… */

// 改变PSO
mCommandList->SetPipelineState(mPSO3.Get());
/* ……使用PSO 3绘制物体…… */
```

换句话说，如果把一个PSO与命令列表相绑定，那么，在我们设置另一个PSO或重置命令列表之前，会一直沿用当前的PSO绘制物体。

注意 Note

考虑到程序的性能问题，我们应当尽可能减少改变PSO状态的次数。为此，若能以一个PSO绘制出所有的物体，绝不用第二个PSO。切记，不要在每次绘制调用时都修改PSO！

6.10 几何图形辅助结构体

在本书中，我们通过创建一个同时存有顶点缓冲区和索引缓冲区的结构体来方便地定义多个几何体。另外，借此结构体即可将顶点和索引数据置于系统内存之中，以供CPU读取。例如，执行拾取（picking）和碰撞检测（collision detection）这样的工作就需要CPU来访问几何体数据。再者，该结构体还缓存了顶点缓冲区和索引缓冲区的一些重要属性（例如格式和每个顶点项所占用的字节数），并提供了返回缓冲区视图的方法。当需要定义多个几何体时，我们就使用下面的`MeshGeometry`（定义于d3dUtil.h头文件中）结构体。

```
// 利用SubmeshGeometry来定义MeshGeometry中存储的单个几何体
// 此结构体适用于将多个几何体数据存于一个顶点缓冲区和一个索引缓冲区的情况
// 它提供了对存于顶点缓冲区和索引缓冲区中的单个几何体进行绘制所需的数据和偏移量，
我们可以据此来
// 实现图6.3中所描绘的技术
struct SubmeshGeometry
{
  UINT IndexCount = 0;
  UINT StartIndexLocation = 0;
  INT BaseVertexLocation = 0;

  // 通过此子网格来定义当前SubmeshGeometry结构体中所存几何体的包围盒（bounding
 box）。我们
  // 将在本书的后续章节中使用此数据
  DirectX::BoundingBox Bounds;
};

struct MeshGeometry
{
  // 指定此几何体网格集合的名称，这样我们就能根据此名找到它
  std::string Name;

  // 系统内存中的副本。由于顶点/索引可以是泛型格式（具体格式依用户而定），所以用B
lob类型来表示
  // 待用户在使用时再将其转换为适当的类型
```

```
  Microsoft::WRL::ComPtr<ID3DBlob> VertexBufferCPU = nullptr;
  Microsoft::WRL::ComPtr<ID3DBlob> IndexBufferCPU = nullptr;

  Microsoft::WRL::ComPtr<ID3D12Resource> VertexBufferGPU = nullptr;
  Microsoft::WRL::ComPtr<ID3D12Resource> IndexBufferGPU = nullptr;

  Microsoft::WRL::ComPtr<ID3D12Resource> VertexBufferUploader = nullptr;
  Microsoft::WRL::ComPtr<ID3D12Resource> IndexBufferUploader = nullptr;

  // 与缓冲区相关的数据
  UINT VertexByteStride = 0;
  UINT VertexBufferByteSize = 0;
  DXGI_FORMAT IndexFormat = DXGI_FORMAT_R16_UINT;
  UINT IndexBufferByteSize = 0;

  // 一个MeshGeometry结构体能够存储一组顶点/索引缓冲区中的多个几何体
  // 若利用下列容器来定义子网格几何体，我们就能单独地绘制出其中的子网格（单个几何
体）
  std::unordered_map<std::string, SubmeshGeometry> DrawArgs;

  D3D12_VERTEX_BUFFER_VIEW VertexBufferView()const
  {
    D3D12_VERTEX_BUFFER_VIEW vbv;
    vbv.BufferLocation = VertexBufferGPU->GetGPUVirtualAddress();
    vbv.StrideInBytes = VertexByteStride;
    vbv.SizeInBytes = VertexBufferByteSize;

    return vbv;
  }

  D3D12_INDEX_BUFFER_VIEW IndexBufferView()const
  {
    D3D12_INDEX_BUFFER_VIEW ibv;
    ibv.BufferLocation = IndexBufferGPU->GetGPUVirtualAddress();
    ibv.Format = IndexFormat;
    ibv.SizeInBytes = IndexBufferByteSize;

    return ibv;
  }

  // 待数据上传至GPU后，我们就能释放这些内存了
  void DisposeUploaders()
  {
    VertexBufferUploader = nullptr;
    IndexBufferUploader = nullptr;
  }
};
```

6.11　立方体演示程序

根据目前所学到的知识，我们足以编写出一个简单的演示程序。所以，我们现在就来渲染一个富有迷幻色彩的立方体（如图6.7所示），以此作为本章的结尾。这个例子其实就是将本章之前讨论的所有内容融汇在一起，提炼为一个单独的程序。读者在研习此程序时，应当配合回顾本章相应的知识点，直到理解每行代码的意义为止。注意，该程序使用的Shaders\color.hlsl文件已在6.5节的末尾处列出。[22]

```
//***************************************************************************
// BoxApp.cpp的作者为Frank Luna (C) 2015 版权所有
//
// 展示如何用Direct3D 12绘制一个立方体
//
// 控制:
//   按下鼠标左键拖动以旋转
//   按下鼠标右键拖动来缩放
//***************************************************************************
#include "../../Common/d3dApp.h"
#include "../../Common/MathHelper.h"
#include "../../Common/UploadBuffer.h"

using Microsoft::WRL::ComPtr;
using namespace DirectX;
using namespace DirectX::PackedVector;

struct Vertex
{
  XMFLOAT3 Pos;
  XMFLOAT4 Color;
};

struct ObjectConstants
{
  XMFLOAT4X4 WorldViewProj = MathHelper::Identity4x4();
};

class BoxApp : public D3DApp
```

```
{
public:
  BoxApp(HINSTANCE hInstance);
  BoxApp(const BoxApp& rhs) = delete;
  BoxApp& operator=(const BoxApp& rhs) = delete;
  ~BoxApp();

  virtual bool Initialize()override;

private:
  virtual void OnResize()override;
  virtual void Update(const GameTimer& gt)override;
  virtual void Draw(const GameTimer& gt)override;

  virtual void OnMouseDown(WPARAM btnState, int x, int y)override;
  virtual void OnMouseUp(WPARAM btnState, int x, int y)override;
  virtual void OnMouseMove(WPARAM btnState, int x, int y)override;

  void BuildDescriptorHeaps();
  void BuildConstantBuffers();
  void BuildRootSignature();
  void BuildShadersAndInputLayout();
  void BuildBoxGeometry();
  void BuildPSO();

private:

  ComPtr mRootSignature = nullptr;
  ComPtr mCbvHeap = nullptr;

  std::unique_ptr> mObjectCB = nullptr;

  std::unique_ptr mBoxGeo = nullptr;

  ComPtr mvsByteCode = nullptr;
  ComPtr mpsByteCode = nullptr;

  std::vector mInputLayout;

  ComPtr mPSO = nullptr;

  XMFLOAT4X4 mWorld = MathHelper::Identity4x4();
  XMFLOAT4X4 mView = MathHelper::Identity4x4();
  XMFLOAT4X4 mProj = MathHelper::Identity4x4();

  float mTheta = 1.5f*XM_PI;
  float mPhi = XM_PIDIV4;
```

```
  float mRadius = 5.0f;

  POINT mLastMousePos;
};

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE prevInstance,
                         PSTR cmdLine, int showCmd)
{
    // 针对调试版本开启运行时内存检测
#if defined(DEBUG) | defined(_DEBUG)
    _CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
#endif

  try
  {
    BoxApp theApp(hInstance);
    if(!theApp.Initialize())
      return 0;

    return theApp.Run();
  }
  catch(DxException& e)
  {
    MessageBox(nullptr, e.ToString().c_str(), L"HR Failed", MB_OK);
    return 0;
  }
}

BoxApp::BoxApp(HINSTANCE hInstance)
: D3DApp(hInstance)
{
}

BoxApp::~BoxApp()
{
}

bool BoxApp::Initialize()
{
  if(!D3DApp::Initialize())
    return false;

  // 重置命令列表为执行初始化命令做好准备工作
  ThrowIfFailed(mCommandList->Reset(mDirectCmdListAlloc.Get(), nullptr));

  BuildDescriptorHeaps();
  BuildConstantBuffers();
```

```
  BuildRootSignature();
  BuildShadersAndInputLayout();
  BuildBoxGeometry();
  BuildPSO();

  // 执行初始化命令
  ThrowIfFailed(mCommandList->Close());
  ID3D12CommandList* cmdsLists[] = { mCommandList.Get() };
  mCommandQueue->ExecuteCommandLists(_countof(cmdsLists), cmdsLists);

  // 等待初始化完成
  FlushCommandQueue();

  return true;
}

void BoxApp::OnResize()
{
  D3DApp::OnResize();

  // 若用户调整了窗口尺寸，则更新纵横比并重新计算投影矩阵
  XMMATRIX P = XMMatrixPerspectiveFovLH(0.25f*MathHelper::Pi,
    AspectRatio(), 1.0f, 1000.0f);
  XMStoreFloat4x4(&mProj, P);
}

void BoxApp::Update(const GameTimer& gt)
{
  // 将球坐标转换为笛卡儿坐标[23]
  float x = mRadius*sinf(mPhi)*cosf(mTheta);
  float z = mRadius*sinf(mPhi)*sinf(mTheta);
  float y = mRadius*cosf(mPhi);

  // 构建观察矩阵
  XMVECTOR pos = XMVectorSet(x, y, z, 1.0f);
  XMVECTOR target = XMVectorZero();
  XMVECTOR up = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);

  XMMATRIX view = XMMatrixLookAtLH(pos, target, up);
  XMStoreFloat4x4(&mView, view);

  XMMATRIX world = XMLoadFloat4x4(&mWorld);
  XMMATRIX proj = XMLoadFloat4x4(&mProj);
  XMMATRIX worldViewProj = world*view*proj;

  // 用当前最新的worldViewProj矩阵来更新常量缓冲区
  ObjectConstants objConstants;
```

```
  XMStoreFloat4x4(&objConstants.WorldViewProj, XMMatrixTranspose(
    worldViewProj));
  mObjectCB->CopyData(0, objConstants);
}

void BoxApp::Draw(const GameTimer& gt)
{
  // 复用记录命令所用的内存
  // 只有当GPU中的命令列表执行完毕后，我们才可对其进行重置
  ThrowIfFailed(mDirectCmdListAlloc->Reset());

  // 通过函数ExecuteCommandList将命令列表加入命令队列后，便可对它进行重置
  // 复用命令列表即复用其相应的内存
  ThrowIfFailed(mCommandList->Reset(mDirectCmdListAlloc.Get(), mPSO.Get())
);

  mCommandList->RSSetViewports(1, &mScreenViewport);
  mCommandList->RSSetScissorRects(1, &mScissorRect);

  // 按照资源的用途指示其状态的转变，此处将资源从呈现状态转换为渲染目标状态
  mCommandList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER::Transition(CurrentBackBuffer(),
    D3D12_RESOURCE_STATE_PRESENT, D3D12_RESOURCE_STATE_RENDER_TARGET));

  // 清除后台缓冲区和深度缓冲区
  mCommandList->ClearRenderTargetView(CurrentBackBufferView(),
    Colors::LightSteelBlue, 0, nullptr);
  mCommandList->ClearDepthStencilView(DepthStencilView(),
    D3D12_CLEAR_FLAG_DEPTH | D3D12_CLEAR_FLAG_STENCIL,
    1.0f, 0, 0, nullptr);

  // 指定将要渲染的目标缓冲区
  mCommandList->OMSetRenderTargets(1, &CurrentBackBufferView(),
    true, &DepthStencilView());

  ID3D12DescriptorHeap* descriptorHeaps[] = { mCbvHeap.Get() };
  mCommandList->SetDescriptorHeaps(_countof(descriptorHeaps), descriptorHe
aps);

  mCommandList->SetGraphicsRootSignature(mRootSignature.Get());

  mCommandList->IASetVertexBuffers(0, 1, &mBoxGeo->VertexBufferView());
  mCommandList->IASetIndexBuffer(&mBoxGeo->IndexBufferView());
  mCommandList->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELI
ST);[24]

  mCommandList->SetGraphicsRootDescriptorTable(
```

```
    0, mCbvHeap->GetGPUDescriptorHandleForHeapStart());

  mCommandList->DrawIndexedInstanced(
    mBoxGeo->DrawArgs["box"].IndexCount,
    1, 0, 0, 0);

  // 按照资源的用途指示其状态的转变，此处将资源从渲染目标状态转换为呈现状态
  mCommandList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER::Transition(CurrentBackBuffer(),
    D3D12_RESOURCE_STATE_RENDER_TARGET, D3D12_RESOURCE_STATE_PRESENT));

  // 完成命令的记录
  ThrowIfFailed(mCommandList->Close());

  // 向命令队列添加欲执行的命令列表
  ID3D12CommandList* cmdsLists[] = { mCommandList.Get() };
  mCommandQueue->ExecuteCommandLists(_countof(cmdsLists), cmdsLists);

  // 交换后台缓冲区与前台缓冲区
  ThrowIfFailed(mSwapChain->Present(0, 0));
  mCurrBackBuffer = (mCurrBackBuffer + 1) % SwapChainBufferCount;

  // 等待绘制此帧的一系列命令执行完毕。这种等待的方法虽然简单却也低效
  // 在后面将展示如何重新组织渲染代码，使我们不必在绘制每一帧时都等待
  FlushCommandQueue();
}

void BoxApp::OnMouseDown(WPARAM btnState, int x, int y)
{
  mLastMousePos.x = x;
  mLastMousePos.y = y;

  SetCapture(mhMainWnd);
}

void BoxApp::OnMouseUp(WPARAM btnState, int x, int y)
{
  ReleaseCapture();
}

void BoxApp::OnMouseMove(WPARAM btnState, int x, int y)
{
  if((btnState & MK_LBUTTON) != 0)
  {
    // 根据鼠标的移动距离计算旋转角度，并令每个像素都按此角度的1/4旋转
    float dx = XMConvertToRadians(0.25f*static_cast
    (x - mLastMousePos.x));
```

```
    float dy = XMConvertToRadians(0.25f*static_cast
    (y - mLastMousePos.y));

    // 根据鼠标的输入来更新摄像机绕立方体旋转的角度
    mTheta += dx;
    mPhi += dy;

    // 限制角度mPhi的范围
    mPhi = MathHelper::Clamp(mPhi, 0.1f, MathHelper::Pi - 0.1f);
  }
  else if((btnState & MK_RBUTTON) != 0)
  {
    // 使场景中的每个像素按鼠标移动距离的0.005倍进行缩放
    float dx = 0.005f*static_cast(x - mLastMousePos.x);
    float dy = 0.005f*static_cast(y - mLastMousePos.y);

    // 根据鼠标的输入更新摄像机的可视范围半径
    mRadius += dx - dy;

    // 限制可视半径的范围
    mRadius = MathHelper::Clamp(mRadius, 3.0f, 15.0f);
  }

  mLastMousePos.x = x;
  mLastMousePos.y = y;
}

void BoxApp::BuildDescriptorHeaps()
{
  D3D12_DESCRIPTOR_HEAP_DESC cbvHeapDesc;
  cbvHeapDesc.NumDescriptors = 1;
  cbvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
  cbvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
    cbvHeapDesc.NodeMask = 0;
  ThrowIfFailed(md3dDevice->CreateDescriptorHeap(&cbvHeapDesc,
    IID_PPV_ARGS(&mCbvHeap)));
}

void BoxApp::BuildConstantBuffers()
{
  mObjectCB = std::make_unique>(md3dDevice.Get(),
    1, true);

  UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(Object-
    Constants));

  D3D12_GPU_VIRTUAL_ADDRESS cbAddress = mObjectCB->Resource()->
```

```
    GetGPUVirtualAddress();
  // 偏移到常量缓冲区中第i个物体所对应的常量数据
  // 这里取i = 0
  int boxCBufIndex = 0;
  cbAddress += boxCBufIndex*objCBByteSize;

  D3D12_CONSTANT_BUFFER_VIEW_DESC cbvDesc;
  cbvDesc.BufferLocation = cbAddress;
  cbvDesc.SizeInBytes = d3dUtil::CalcConstantBufferByteSize(sizeof(Object-
    Constants));

  md3dDevice->CreateConstantBufferView(
    &cbvDesc,
    mCbvHeap->GetCPUDescriptorHandleForHeapStart());
}

void BoxApp::BuildRootSignature()
{
  // 着色器程序一般需要以资源作为输入（例如常量缓冲区、纹理、采样器等）
  // 根签名则定义了着色器程序所需的具体资源
  // 如果把着色器程序看作一个函数，而将输入的资源当作向函数传递的参数数据，那么便
可类似地认为根签名
  // 定义的是函数签名

  // 根参数可以是描述符表、根描述符或根常量
  CD3DX12_ROOT_PARAMETER slotRootParameter[1];

  // 创建由单个CBV所组成的描述符表
  CD3DX12_DESCRIPTOR_RANGE cbvTable;
  cbvTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 0);
  slotRootParameter[0].InitAsDescriptorTable(1, &cbvTable);

  // 根签名由一组根参数构成
  CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(1, slotRootParameter, 0, nullptr
,
    D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

  // 用单个寄存器槽来创建一个根签名，该槽位指向一个仅含有单个常量缓冲区的描述符区
域
  ComPtr serializedRootSig = nullptr;
  ComPtr errorBlob = nullptr;
  HRESULT hr = D3D12SerializeRootSignature(&rootSigDesc, D3D_ROOT_SIGNATUR
E_VERSION_1,
  serializedRootSig.GetAddressOf(),
     errorBlob.GetAddressOf());

  if(errorBlob != nullptr)
```

```
  {
    ::OutputDebugStringA((char*)errorBlob->GetBufferPointer());
  }
  ThrowIfFailed(hr);

  ThrowIfFailed(md3dDevice->CreateRootSignature(
    0,
    serializedRootSig->GetBufferPointer(),
    serializedRootSig->GetBufferSize(),
    IID_PPV_ARGS(&mRootSignature)));
}

void BoxApp::BuildShadersAndInputLayout()
{
  HRESULT hr = S_OK;

  mvsByteCode = d3dUtil::CompileShader(L"Shaders\\color.hlsl", nullptr,
    "VS", "vs_5_0");
  mpsByteCode = d3dUtil::CompileShader(L"Shaders\\color.hlsl", nullptr,
    "PS", "ps_5_0");

  mInputLayout =
  {
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
      D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
    { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
      D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }
  };
}

void BoxApp::BuildBoxGeometry()
{
  std::array vertices =
  {
    Vertex({ XMFLOAT3(-1.0f, -1.0f, -1.0f), XMFLOAT4(Colors::White) }),
    Vertex({ XMFLOAT3(-1.0f, +1.0f, -1.0f), XMFLOAT4(Colors::Black) }),
    Vertex({ XMFLOAT3(+1.0f, +1.0f, -1.0f), XMFLOAT4(Colors::Red) }),
    Vertex({ XMFLOAT3(+1.0f, -1.0f, -1.0f), XMFLOAT4(Colors::Green) }),
    Vertex({ XMFLOAT3(-1.0f, -1.0f, +1.0f), XMFLOAT4(Colors::Blue) }),
    Vertex({ XMFLOAT3(-1.0f, +1.0f, +1.0f), XMFLOAT4(Colors::Yellow) }),
    Vertex({ XMFLOAT3(+1.0f, +1.0f, +1.0f), XMFLOAT4(Colors::Cyan) }),
    Vertex({ XMFLOAT3(+1.0f, -1.0f, +1.0f), XMFLOAT4(Colors::Magenta) })
  };

  std::array indices =
  {
    // 立方体前表面
```

```
  0, 1, 2,
  0, 2, 3,

  // 立方体后表面
  4, 6, 5,
  4, 7, 6,

  // 立方体左表面
  4, 5, 1,
  4, 1, 0,

  // 立方体右表面
  3, 2, 6,
  3, 6, 7,

  // 立方体上表面
  1, 5, 6,
  1, 6, 2,

  // 立方体下表面
  4, 0, 3,
  4, 3, 7
};

const UINT vbByteSize = (UINT)vertices.size() * sizeof(Vertex);
const UINT ibByteSize = (UINT)indices.size() * sizeof(std::uint16_t);

mBoxGeo = std::make_unique();
mBoxGeo->Name = "boxGeo";

ThrowIfFailed(D3DCreateBlob(vbByteSize, &mBoxGeo->VertexBufferCPU));
CopyMemory(mBoxGeo->VertexBufferCPU->GetBufferPointer(),
  vertices.data(), vbByteSize);

ThrowIfFailed(D3DCreateBlob(ibByteSize, &mBoxGeo->IndexBufferCPU));
CopyMemory(mBoxGeo->IndexBufferCPU->GetBufferPointer(),
  indices.data(), ibByteSize);

mBoxGeo->VertexBufferGPU = d3dUtil::CreateDefaultBuffer(
  md3dDevice.Get(), mCommandList.Get(),
  vertices.data(), vbByteSize,
  mBoxGeo->VertexBufferUploader);

mBoxGeo->IndexBufferGPU = d3dUtil::CreateDefaultBuffer(
  md3dDevice.Get(), mCommandList.Get(),
  indices.data(), ibByteSize,
  mBoxGeo->IndexBufferUploader);
```

```
  mBoxGeo->VertexByteStride = sizeof(Vertex);
  mBoxGeo->VertexBufferByteSize = vbByteSize;
  mBoxGeo->IndexFormat = DXGI_FORMAT_R16_UINT;
  mBoxGeo->IndexBufferByteSize = ibByteSize;

  SubmeshGeometry submesh;
  submesh.IndexCount = (UINT)indices.size();
  submesh.StartIndexLocation = 0;
  submesh.BaseVertexLocation = 0;

  mBoxGeo->DrawArgs["box"] = submesh;
}

void BoxApp::BuildPSO()
{
  D3D12_GRAPHICS_PIPELINE_STATE_DESC psoDesc;
  ZeroMemory(&psoDesc, sizeof(D3D12_GRAPHICS_PIPELINE_STATE_DESC));
  psoDesc.InputLayout = { mInputLayout.data(), (UINT)mInputLayout.size() }
;
  psoDesc.pRootSignature = mRootSignature.Get();
  psoDesc.VS =
    {
        reinterpret_cast(mvsByteCode->GetBufferPointer()),
        mvsByteCode->GetBufferSize()
    };
  psoDesc.PS =
    {
        reinterpret_cast(mpsByteCode->GetBufferPointer()),
        mpsByteCode->GetBufferSize()
    };
  psoDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);
  psoDesc.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT);
  psoDesc.DepthStencilState = CD3DX12_DEPTH_STENCIL_DESC(D3D12_DEFAULT);
  psoDesc.SampleMask = UINT_MAX;
  psoDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE;
  psoDesc.NumRenderTargets = 1;
  psoDesc.RTVFormats[0] = mBackBufferFormat;
  psoDesc.SampleDesc.Count = m4xMsaaState ? 4 : 1;
  psoDesc.SampleDesc.Quality = m4xMsaaState ? (m4xMsaaQuality - 1) : 0;
  psoDesc.DSVFormat = mDepthStencilFormat;
  ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&psoDesc,
    IID_PPV_ARGS(&mPSO)));
}
```

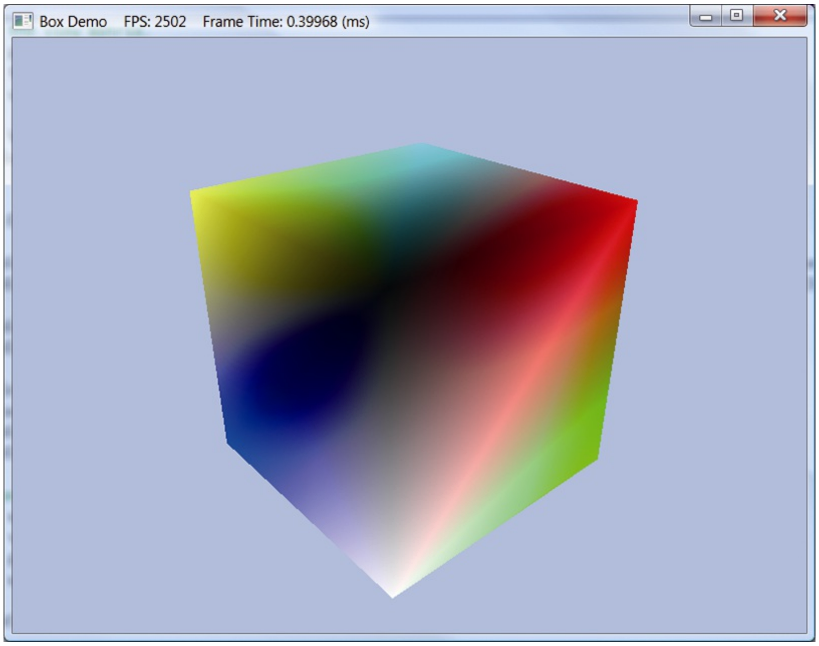


图6.7 “立方体”例程示意图

6.12　小结

1．除了空间位置信息，Direct3D中的顶点还可以存储其他类型的属性数据。为了创建自定义的顶点格式，我们首先要将选定的顶点数据定义为一个结构体。待顶点结构体定义好后，便可用输入布局描述（D3D12_INPUT_LAYOUT_DESC）向Direct3D提供其细节。输入布局描述由两部分组成：一个D3D12_INPUT_ELEMENT_DESC元素构成的数组，一个记录该数组中元素个数的无符号整数。D3D12_INPUT_ELEMENT_DESC数组中的元素要与顶点结构体中的成员一一对应。事实上，输入布局描述为结构体D3D12_GRAPHICS_PIPELINE_STATE_DESC中的一个字段，这就是说，它实为PSO的一个组成部分，用于与顶点着色器输入签名进行格式的比对验证。因此，当PSO与渲染流水线绑定时，输入布局也将以PSO组成元素的身份随PSO与渲染流水线的IA阶段相绑定。

2．为了使GPU可以访问顶点/索引数组，便需要将其置于一种名为缓冲区（buffer）的资源之内。顶点数据与索引数据的缓冲区分别称为顶点缓冲区（vertex buffer）和索引缓冲区（index buffer）。缓冲区用接口ID3D12Resource表示，要创建缓冲区资源需要填写D3D12_RESOURCE_DESC结构体，再调用ID3D12Device::CreateCommittedResource方法。顶点缓冲区的视图与索引缓冲区的视图分别用D3D12_VERTEX_BUFFER_VIEW与D3D12_INDEX_BUFFER_VIEW结构体加以描述。随后，即可通过

ID3D12GraphicsCommandList::IASetVertexBuffers方法与ID3D12GraphicsCommandList::IASetIndexBuffer方法分别将顶点缓冲区与索引缓冲区绑定到渲染流水线的IA阶段。最后，要绘制非索引（non-indexed）描述的几何体（即以顶点数据来绘制的几何体）可借助ID3D12GraphicsCommandList::DrawInstanced方法，而以索引描述的几何体可由ID3D12GraphicsCommandList::DrawIndexedInstanced方法进行绘制。

3．顶点着色器是一种用HLSL编写并在GPU上运行的程序，它以单个顶点作为输入与输出。每个待绘制的顶点都要流经顶点着色器阶段。这使得程序员能够在此以顶点为基本单位进行处理，继而获取多种多样的渲染效果。从顶点着色器输出的数据将传至渲染流水线的下一个阶段。

4．常量缓冲区是一种GPU资源（ID3D12Resource），其数据内容可供着色器程序引用。它们被创建在上传堆（upload heap）而非默认堆（default heap）中。因此，应用程序可通过将数据从系统内存复制到显存中来更新常量缓冲区。如此一来，C++应用程序就可与着色器通信，并更新常量缓冲区内着色器所需的数据。例如，C++程序可以借助这种方式对着色器所用的“世界—观察—投影”矩阵进行更改。在此，我们建议读者考量数据更新的频繁程度，以此为依据来创建不同的常量缓冲区。效率乃是划分常量缓冲区的动机。在对一个常量缓冲区进行更新的时候，其中的所有变量都会随之更新，正所谓牵一发而动全身。因此，

应根据更新频率将数据有效地组织为不同的常量缓冲区，以此来避免无谓的冗余的更新，从而提高效率。

5．像素着色器是一种用HLSL编写且运行在GPU上的程序，它以经过插值计算所得到的顶点数据作为输入，待处理后，再输出与之对应的一种颜色值。由于硬件优化的原因，某些像素片段可能还未到像素着色器就已被渲染流水线剔除了（例如采用了提前深度剔除技术，early-z rejection）。像素着色器可使程序员以像素为基本单位进行处理，从而获得变化万千的渲染效果。从像素着色器输出的数据将被移交至渲染流水线的下一个阶段。

6．大多数控制图形流水线状态的Direct3D对象都被指定到了一种称作流水线状态对象（pipeline state object，PSO）的集合之中，并用`ID3D12PipelineState`接口来表示。我们将这些对象集总起来再统一对渲染流水线进行设置，是出于对性能因素的考虑。这样一来，Direct3D就能验证所有的状态是否彼此兼容，而驱动程序也将可以提前生成硬件本地指令及其状态。

6.13 练习

1．写出与下列顶点结构体所对应的D3D12_INPUT_ELEMENT_DESC数组：

```
struct Vertex
{
    XMFLOAT3 Pos;
    XMFLOAT3 Tangent;
    XMFLOAT3 Normal;
    XMFLOAT2 Tex0;
    XMFLOAT2 Tex1;
    XMCOLOR Color;
};
```

2．改写彩色立方体演示程序，这次使用两个顶点缓冲区（以及两个输入槽）来向渲染流水线传送顶点数据。这两个顶点缓冲区，一个用来存储位置元素，另一个用来储存颜色元素。此时，我们应当利用两个顶点结构体以下列方式分别存放这两种不同的数据：

```
struct VPosData
{
  XMFLOAT3 Pos;
};

struct VColorData
{
  XMFLOAT4 Color;
};
```

我们编写的D3D12_INPUT_ELEMENT_DESC数组应当是这样子的：

```
D3D12_INPUT_ELEMENT_DESC vertexDesc[] =
{
  {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
```

```
      D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
    {"COLOR",  0, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 0,
      D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0}
  };
```

此后，位置元素会被连接到输入槽0，而颜色元素将被连接到输入槽1。而且我们还可以发现，对于这两种元素来说，它们的`D3D12_INPUT_ELEMENT_DESC::AlignedByteOffset`字段皆为0。这是因为位置和颜色这两种元素不再共用输入槽，而是独享各自的输入槽。接下来，我们再用`ID3D12GraphicsCommandList::IASetVertexBuffers`方法，将两个顶点缓冲区分别与寄存器槽0和槽1相绑定。自此，Direct3D也将用来自不同输入槽的元素来装配顶点（合成顶点数据）。这种将元素分槽存放的方法也可用于程序优化。例如，在实现阴影贴图算法（shadow mapping algorithm，或称阴影映射算法）的过程中，我们需要在每一帧绘制场景两次：第一次是从光源的视角渲染场景（shadow pass，阴影绘制过程[25]），第二次是从主摄像机的视角来渲染场景（main pass，主渲染过程）。阴影绘制过程只需要用到位置数据和纹理坐标（为了对几何体进行alpha测试）。所以，我们可以将顶点数据分到两个槽中：一个槽容纳位置信息和纹理坐标，另一个槽存储顶点的其他属性（如法向量和切向量）。如此一来，阴影绘制过程所需的顶点数据只用一个数据流即可（存有位置信息和纹理坐标的那组数据），从而节省阴影绘制过程所消耗的数据带宽。而主渲染过程将同时使用这两个顶点输入槽，以获得所需的全部顶点数据。考虑到程序性能，建议您尽量减少输入槽的使用数量，最好不要多于3个。

3．绘制：

（a）图5.13a中所示的点列表。

（b）图5.13b中所示的线条带。

（c）图5.13c中所示的线列表。

（d）图5.13d中所示的三角形带。

（e）图5.14a中所示的三角形列表。

4．构造图6.8所示的金字塔的顶点列表和索引列表，并将其绘制出来：令塔顶为红色，其他底座顶点为绿色。

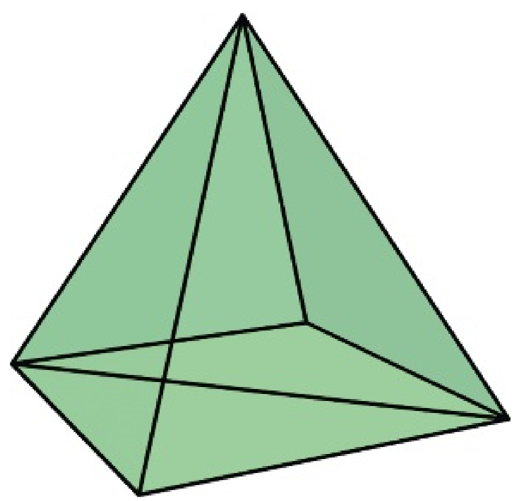


图6.8　构成四棱锥的三角形

5．查阅本章的“Box”（立方体）演示程序代码可知，我们只对立方体的顶点处指定了颜色。运行此程序后会发现立方体却处处都呈现出了不同的色彩，那么问题就来了：构成立方体表面的三角形内的像素是怎

样得到各自像素颜色的呢？

6．修改“Box”演示程序，在顶点着色器将诸顶点变换到世界空间之前，先对顶点应用下列变换。

```
vin.PosL.xy += 0.5f*sin(vin.PosL.x)*sin(3.0f*gTime);
vin.PosL.z *= 0.6f + 0.4f*sin(2.0f*gTime);
```

为此，我们需要在程序中添加一个常量缓冲区变量gTime，此变量为函数GameTimer::TotalTime()的当前值。这段代码将通过时间函数驱动各顶点，使立方体的形状随正弦函数周期性地发生改变。

7．将立方体和练习4中金字塔的顶点合并到一个大的顶点缓冲区内，再将两者的索引合并至一个大的索引缓冲区中（但是不要更新索引值）。接着，再根据上述改动来依次调整调用ID3D12GraphicsCommandList::DrawIndexedInstanced方法的参数，以正确地绘制立方体和金字塔。并通过设置世界变换矩阵，使两者在世界空间中互不相交。

8．修改“Box”演示程序，以线框模式来渲染立方体。

9．修改“Box”演示程序，首先禁用背面剔除（D3D12_CULL_MODE_NONE）并运行程序；随后，再代以正面剔除（D3D12_CULL_MODE_FRONT）试之。用线框模式输出程序的绘图效果，可便于我们观察不同剔除模式之间的绘制差别。

10．如果在某些情况下需要缩减顶点占用内存的大小，那么不妨将

颜色的精度从128位减少到32位。修改“Box”演示程序，将其顶点结构体中128位的颜色值调整为32位颜色值。此时，顶点结构体及其对应的顶点输入描述将变为：

```
struct Vertex
{
  XMFLOAT3 Pos;
  XMCOLOR Color;
};

D3D12_INPUT_ELEMENT_DESC vertexDesc[] =
{
  {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
  {"COLOR",  0, DXGI_FORMAT_B8G8R8A8_UNORM, 0, 12,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0}
}
```

这时，我们应使用DXGI_FORMAT_B8G8R8A8_UNORM格式（8位蓝色、8位绿色、8位红色以及8位alpha值）来表示颜色元素。它对应于常用的32位图像颜色格式ARGB，但颜色通道的顺序是否看起来有点奇怪？这是因为DXGI_FORMAT符号所列的值在内存中是用小端（little-endian）字节序来表示的。在小端字节序表示法中，多字节的数据字（data word）是从最低有效字节开始写至最高有效字节的，这就是为什么格式ARGB在内存中被表示为BGRA；它的最低有效字节处于最低内存地址，而最高有效字节则列于最高内存地址。

11．考虑下列C++顶点结构体：

```
struct Vertex
{
  XMFLOAT3 Pos;
  XMFLOAT4 Color;
};
```

（a）输入布局描述中的元素顺序需要与顶点结构体中的元素顺序相匹配吗？即下列顶点声明对于上述顶点结构体来说是否恰当呢？通过实验来寻找答案，并给出它能工作或出错的原因。

```
D3D12_INPUT_ELEMENT_DESC vertexDesc[] =
{
  {"COLOR",  0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
  {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
};
```

（b）顶点着色器结构体中的元素顺序是否需要与C++顶点结构体中的元素顺序相匹配呢？即下列顶点着色器结构体是否能与上述C++顶点结构体一起协同工作呢？通过实验来寻求答案，并给出它能否正常工作的原因。

```
struct VertexIn
{
  float4 Color : COLOR;
  float3 Pos  : POSITION;
};
```

12．设置视口，使它对准后台缓冲区的左半部分。

13．借助裁剪测试，剔除后台缓冲区中心宽为mClientWidth/2、高为mClientHeight/2这一矩形范围之外的所有像素。注意，要做到这一点，我们还需以D3D12_RECT结构体描述裁剪范围，并调用RSSetScissorRects方法对此进行设定。

14．用像素着色器实现出可变色立方体的效果。在顶点着色器和像

素着色器中，通过使用常量缓冲区以及简单的控制函数，使立方体的颜色随着时间的推移而平滑地发生变化。

15．修改“Box”演示程序中的像素着色器如下：

```
float4 PS(VertexOut pin) : SV_Target
{
  clip(pin.Color.r - 0.5f);
  return pin.Color;
}
```

运行该示例，并猜想内置函数clip起到了什么作用。

16．修改“Box”演示程序中的像素着色器，根据插值顶点颜色与在常量缓冲区中指定的颜色gPulseColor，计算出两者间平滑的过渡颜色。为了实现这个目标，我们需要在应用程序端更新此常量缓冲区。修改后的HLSL代码中的常量缓冲区和像素着色器应当如下：

```
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorldViewProj;
  float4 gPulseColor;
  float gTime;
};

float4 PS(VertexOut pin) : SV_Target
{
  const float pi = 3.14159;

  // 随着时间流逝，令正弦函数的值在[0,1]区间内周期性地变化
  float s = 0.5f*sin(2*gTime - 0.25f*pi)+0.5f;

  // 基于参数s 在pin.Color与gPulseColor之间进行线性插值
  float4 c = lerp(pin.Color, gPulseColor, s);

  return c;
}
```

其中的变量gTime对应于函数GameTimer::TotalTime()给出的当

前值。

---

[1]　此处提及的“签名（signature）”即着色器的输入或输出参数列表。

[2]　预览版中为D3D12_INPUT_PER_VERTEX_DATA。

[3]　预览版中为D3D12_INPUT_PER_INSTANCE_DATA。

[4]　在正式版中，ID3D12CommandList::CopySubresourceRegion（这是预览版中的函数名）函数已细分为主要负责复制纹理的函数CopyTextureRegion（即也可复制缓冲区资源），与负责复制缓冲区数据的函数CopyBufferRegion，它们皆继承自ID3D12GraphicsCommandList接口。而且UpdateSubresources函数亦有修改，此函数共有3种实现，第一种以一块分配的内存真正地实现了将资源从上传堆复制到默认堆，而另外两种则分别先在堆或栈中分配复制过程中所需的内存，再调用第一种实现去执行实际的复制操作。最后，上传堆内的数据最终是被复制到以CD3DX12_TEXTURE_COPY_LOCATION Dst表示的默认堆中，而不是mBuffer中。

[5]　在预览版中，其名为D3D12_VERTEX_BUFFER_DESC。

[6]　Direct3D 11提供了多达7种绘制调用方法，而Direct3D 12则将其精简为3种，且都为实例化绘制方法。文中介绍的是采用顶点来绘制几何

体的方法，另一种 `ID3D12GraphicsCommandList::DrawIndexedInstanced`为需要提供索引数据的绘制方法，后文会讲到。

[7] 希望更为深入地学习HLSL语言，可以访问GitHub网站。它基本上代表了HLSL着色模型（Shader Model，SM）的最新动向。如果想要更进一步，还可以尝试阅读《Asm Shader Reference》（bb219840，尽管现在的Direct3D版本事实上已不再支持直接编写汇编代码了，不过在程序的优化方面可能会有帮助）。

[8] 系统值语义是在Direct3D 10引入的。Direct3D 10及其后续版本中的`SV_Position`语义，与Direct3D 9中的`POSITION`语义等价。其它语义的对照关系与使用方法请参考《Semantics》（bb509647）。

[9] 简而言之，在顶点着色器运行之前，每个顶点中的数据都会载入（顶点着色器的）输入寄存器内，供其执行期间使用。输入布局中的格式定义的是：数据进入输入寄存器之前的类型（具体来讲，输入布局描述的是：在渲染流水线中顶点着色器之前的输入装配器阶段内，输入缓冲区（即顶点缓冲区与索引缓冲区）的数据类型）。而输入签名定义的是：在执行顶点着色器程序的过程中，将数据从输入寄存器中读取时所视作的类型。由于二者中存在元素格式相异的情况，在编译期间才触发了这一警告。如果是程序员故意而为之则可视而不见，否则需要使二者相匹配。例如，大家可以尝试按上面元素格式不匹配的方式进行绘制，看看最终效果。再想一想为什么会出现这种情况，利用前文中提到的调试工具或其他手段验证一下。

[10] 这里所提到的“像素”是最终写入后台缓冲区中数据，而“像素片段”是写入此“像素”过程中的竞争者。与前文中讲采样时的术语是两回事。

[11] 着色器模型定义了HLSL的编写规范，确定了其内置函数（HLSL intrinsic functions）、着色器属性等一切语言元素。虽然Direct3D 11.3与Direct3D 12的发行时间相近，且部分官方文档宣称“Direct3D 11.3支持着色器模型5.1”，但实际并非如此。着色器模型5.1与6.0为Direct3D 12所独有。我们也可以通过DirectX所提供的工具dxcapsviewer.exe（位于“系统盘:\Program Files (x86)\Windows Kits\10\bin\”下的文件夹）来验证这一点。多说一句，就这一点也可以从侧面看出，微软公司并没有抛弃Direct3D 11，甚至在后面继续将其更新到11.4，使之与Direct3D 12的功能更加接近，并令二者（乃至Direct3D 10与Direct2D）之间可以进行互操作（interop）。另外，前文中也提到了微软开源的着色器编译器，它目前已支持SM 6.2，并附加了一些实用工具。

[12] 关于此处的子资源索引，请参考12.3.4节。

[13] Direct3D 12不仅保证了`map`与`unmap`函数在多线程中调用的安全性，还令`map`函数可嵌套调用。第一次调用`map`函数时，Direct3D会在CPU端分配一块虚拟内存地址范围，用来映射GPU中的资源。而最后一次调用`unmap`函数时，则会释放这块CPU虚拟地址范围。`map`函数会在必要时对CPU缓存执行invalidate操作（标记相关缓存无效，令CPU读取主存中的数据），以此使CPU端可以读取GPU端对这段地址内容所做的修改；相反地，`unmap`函数则会在必要时对CPU缓存进行flush操作（令

缓存中的数据写回主存），以令GPU端可以读取CPU端对这段地址内容所做的修改。

[14]　寄存器槽就是向着色器传递资源的手段，register(*#)中*表示寄存器传递的资源类型，可以是t（表示着色器资源视图）、s（采样器）、u（无序访问视图）以及b（常量缓冲区视图），#则为所用的寄存器编号。

[15]　Direct3D 12规定，必须先将根签名的描述布局进行序列化处理（serialize），待其转换为以`ID3DBlob`接口表示的序列化数据格式后，才可将它传入`CreateRootSignature`方法，正式创建根签名。在此，可以设置将根签名按何种版本（1.0，1.1）进行序列化处理。在Windows 10（14393）版以后，可以`D3D12SerializeVersionedRootSignature`方法代之。

[16]　自Windows 10周年更新版的SDK（Windows 10 Anniversary Update SDK，1607）起，编译器会默认将根签名按版本1.1进行编译，用户可通过对编译器进行配置使它创建1.0版本的根签名（详见Root Signature Version 1.1中的Version management部分，mt709473）。根签名1.1版相对于1.0版而言会把描述符与数据分为static与volatile两种，从而使图形驱动层的行为得到优化。而且，在不支持根签名1.1版本的操作系统上使用1.1版本的根签名会出现问题。

[17]　目前最新的着色器模型版本是6.4，需用前文注释中的DirectX Shader Compiler进行编译。着色器模型6.0～6.4添加了许多新的内置函

数与数据类型。读者可以从msdn与微软GitHub上DirectXShaderCompiler项目的示例及wiki文档中获得更多相关信息（使用该版本着色器模型有SDK版本与特性级别等限制）。

[18]　如之前所注，CD3D12_DEFAULT在后续的SDK版本中已更名为CD3DX12_DEFAULT。

[19]　在预览版中，D3D12_FILL_MODE中的枚举项以D3D12_FILL_*作为前缀，且D3D12_CULL_MODE中的枚举项以D3D12_CULL_*作为前缀，而CD3DX12_DEFAULT则名为CD3D12_DEFAULT。

[20]　文中并没有对D3D12_GRAPHICS_PIPELINE_STATE_DESC结构体中的参数完全介绍，全书也未曾用到这些参数，可能是作者因此而未写。

[21]　如之前所注，代码中的CD3DX12_RASTERIZER_DESC、CD3DX12_BLEND_DESC与CD3DX12_DEPTH_STENCIL_DESC等辅助方法在预览SDK版本中为CD3D12前缀。

[22]　在Windows 10中，图形调试工具已改为按可选安装的形式出现。在Windows 10环境下第一次运行Direct3D调试版程序，可能会得到相关的错误提示。这时可在设置/系统/应用和功能/管理可选功能中（视具体系统版本而定）找到“图形工具”安装。有时也会发生找不到此安装项的情况。此时，可在管理员身份的命令行工具（cmd）中用命令Dism /online /add-capability /capabilityname:Tools.Graphics.DirectX~~~~0.0.1.0

来安装。也可在微软官网下载相应的HLK补充测试包（HLK_GRFX_FOD.zip，注意与自己的系统版本匹配）进行离线安装。

[23]　由于摄像机涉及环绕物体旋转等操作，所以先利用球坐标表示变换（可将摄像机视为针对目标物体的可控“侦查卫星”，用鼠标调整摄像机与物体间的距离（也就是球面半径）以及观察角度），再将其转换为笛卡儿坐标的表示更为方便。而这一摄像机系统也被称为旋转摄像机系统（orbiting camera system，也有作环绕摄像机系统）。顺便提一句，除了上述两种坐标系之外，圆柱坐标系（cylindrical coordinate system，也有作柱面坐标系）有时也会派上用场，因此掌握这3种坐标系间的转换对实际应用也会大有裨益。

[24]　这里使用`D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST`对于运行结果无碍，但是为了程序的统一性，最好将其设置为`D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST`。

[25]　在早期的Direct3D版本中，存在一种效果框架（effects framework）。用户借助效果文件（effect file）即可实现所需的渲染效果。效果文件中存有一个或多个technique，一个technique就是实现某种特效的具体处理手段（即一个technique对应一种特效），每个technique都由一个或多个pass构成（即该technique的实现由其下这些pass来具体执行）。a (rendering) pass一词常被称作“一趟”“一通”或“一遍”绘制过程（也有译作绘制路径、渲染通道等）。一个pass则含有绘制过程中所需的各种渲染状态以及着色器等。借助多pass即可渲染出用户所需的各种特效。虽然这一框架在Direct3D 12中已不复存在了，但是借助多pass实

现特效这一技术手段却依然适用。

第7章　利用Direct3D绘制几何体（续）

本章将介绍一些此书后面常会用到的绘图模式。首先讲解与绘图优化相关的内容，此处涉及“帧资源（frame resources）”等概念。若采用帧资源，我们就得修改程序中的渲染循环，这样做所带来的好处是不必在每一帧都刷新命令队列，继而改善CPU和GPU的利用率。接下来，我们会提出渲染项（render item）的概念，并解释如何基于更新频率来划分常量数据。此外，我们将研究根签名的更多细节，并学习其他两种根参数类型：根描述符和根常量。最后，我们还会展示怎样绘制更为复杂的物体。完成本章的学习后，读者将能够绘制出形如山川的表面，还有圆台、球体以及模拟波浪运动的动画。

学习目标：

1．学会一种无须每帧都要刷新命令队列的渲染流程，由此来优化程序的性能。

2．了解另外两种根签名参数类型：根描述符和根常量。

3．探索如何在程序中生成和绘制常见的几何体，如栅格、圆台和球体。

4．研究怎样通过动态顶点缓冲区来更新CPU中的顶点数据，并且向GPU上传顶点的新位置信息。

7.1 帧资源

首先回顾一下4.2节中描述的CPU与GPU并行工作的情形：CPU（除了要执行其他必要的工作之外，还要）构建并提交命令列表，而GPU则负责处理命令队列中的各种命令。我们的目标是令CPU和GPU持续工作，从而充分利用系统当中的可用硬件资源。到目前为止，我们的演示程序在绘制每一帧时都会将CPU和GPU进行一次同步。这样做的原因有两个：

1．在GPU未结束命令分配器（command allocator）中所有命令的执行之前，不能将它重置。如若不进行同步，那么在GPU完成当前第n帧的处理之前，CPU可能会继续执行下一帧（第$n+1$帧）的相关工作：如果CPU在第$n+1$帧中重置了命令分配器，那么，GPU当前还未处理的命令就会被清除掉。

2．在GPU未完成与常量缓冲区相关的绘制命令之前，CPU不可更新这些常量缓冲区。这种情景在4.2.2小节和图4.7中有相应的描述。假设我们不进行同步，那么在GPU结束当前第n帧的处理之前，CPU可能会继续执行第$n+1$帧的相关工作：如果CPU在第$n+1$帧中覆写了常量缓冲区内的数据，而GPU还未曾引用第n帧中的常量缓冲区数据去执行绘制调用，那么，在GPU正式绘制第n帧画面时，常量缓冲区内所存的将不是此帧所需的数据。

所以，我们在每帧绘制的结尾都会调

用`D3DApp::FlushCommandQueue`函数，以确保GPU在每一帧都能正确完成所有命令的执行。这种解决方案虽然奏效却效率低下，原因如下：

1．在每帧的起始阶段，GPU不会执行任何命令，因为等待它处理的命令队列空空如也。这种情况将持续到CPU构建并提交一些供GPU执行的命令为止。

2．在每帧的收尾阶段，CPU会等待GPU完成命令的处理。

所以，CPU和GPU在每一帧都存在各自的空闲时间。

解决此问题的一种方案是：以CPU每帧都需更新的资源作为基本元素，创建一个环形数组（circular array，也有译作循环数组）。我们称这些资源为帧资源（frame resource），而这种循环数组通常是由3个帧资源元素所构成的。该方案的思路是：在处理第n帧的时候，CPU将周而复始地从帧资源数组中获取下一个可用的（即没被GPU使用中的）帧资源。趁着GPU还在处理此前帧之时，CPU将为第n帧更新资源，并构建和提交对应的命令列表。随后，CPU会继续针对第$n+1$帧执行同样的工作流程，并不断重复下去。如果帧资源数组共有3个元素，则令CPU比GPU提前处理两帧，以确保GPU可持续工作。下面所列的是帧资源类的例程，在本章中我们将利用“Shapes”（不同形状的几何体）程序配合演示。由于在此例中CPU只需修改常量缓冲区，所以程序中的帧资源类只含有常量缓冲区。

```
// 存有CPU为构建每帧命令列表所需的资源
// 其中的数据将依程序而异，这取决于实际绘制所需的资源
struct FrameResource
{
```

```
public:

  FrameResource(ID3D12Device* device, UINT passCount, UINT objectCount);
  FrameResource(const FrameResource& rhs) = delete;
  FrameResource& operator=(const FrameResource& rhs) = delete;
  ~FrameResource();

  // 在GPU处理完与此命令分配器相关的命令之前，我们不能对它进行重置。
  // 所以每一帧都要有它们自己的命令分配器
  Microsoft::WRL::ComPtr<ID3D12CommandAllocator> CmdListAlloc;

  // 在GPU执行完引用此常量缓冲区的命令之前，我们不能对它进行更新。
  // 因此每一帧都要有它们自己的常量缓冲区
  std::unique_ptr<UploadBuffer<PassConstants>> PassCB = nullptr;
  std::unique_ptr<UploadBuffer<ObjectConstants>> ObjectCB = nullptr;

  // 通过围栏值将命令标记到此围栏点，这使我们可以检测到GPU是否还在使用这些帧资源
  UINT64 Fence = 0;
};

FrameResource::FrameResource(ID3D12Device* device, UINT passCount, UINT
  objectCount)
{
  ThrowIfFailed(device->CreateCommandAllocator(
    D3D12_COMMAND_LIST_TYPE_DIRECT,
    IID_PPV_ARGS(CmdListAlloc.GetAddressOf())));

  PassCB = std::make_unique<UploadBuffer<PassConstants>>(device,
    passCount, true);
  ObjectCB = std::make_unique<UploadBuffer<ObjectConstants>>(device,
    objectCount, true);
}
FrameResource::~FrameResource() { }
```

据此，我们的应用程序类（ShapesApp）将实例化一个由3个帧资源元素所构成的向量，并留有特定的成员变量来记录当前的帧资源：

```
const int gNumFrameResources = 3;
std::vector<std::unique_ptr<FrameResource>> mFrameResources;
FrameResource* mCurrFrameResource = nullptr;
int mCurrFrameResourceIndex = 0;

void ShapesApp::BuildFrameResources()
{
  for(int i = 0; i < gNumFrameResources; ++i)
```

```
  {
    mFrameResources.push_back(std::make_unique<FrameResource>(
      md3dDevice.Get(), 1, (UINT)mAllRitems.size()));
  }
}
```

现在，CPU端处理第n帧的算法是这样的：

```
void ShapesApp::Update(const GameTimer& gt)
{
  // 循环往复地获取帧资源循环数组中的元素
  mCurrFrameResourceIndex = (mCurrFrameResourceIndex + 1) % gNumFrameResou
rces;
  mCurrFrameResource = mFrameResources[mCurrFrameResourceIndex].get();

  // GPU端是否已经执行完处理当前帧资源的所有命令呢？
  // 如果还没有就令CPU等待，直到GPU完成命令的执行并抵达这个围栏点
  if(mCurrFrameResource->Fence != 0 &&
    mCommandQueue->GetLastCompletedFence() < mCurrFrameResource->Fence)
  {
    HANDLE eventHandle = CreateEventEx(nullptr, false, false, EVENT_ALL_AC
CESS);
    ThrowIfFailed(mCommandQueue->SetEventOnFenceCompletion(
      mCurrFrameResource->Fence, eventHandle));[1]
    WaitForSingleObject(eventHandle, INFINITE);
    CloseHandle(eventHandle);
  }

  // [...] 更新mCurrFrameResource内的资源（例如常量缓冲区）
}

void ShapesApp::Draw(const GameTimer& gt)
{
  // [...] 构建和提交本帧的命令列表

  // 增加围栏值，将命令标记到此围栏点
  mCurrFrameResource->Fence = ++mCurrentFence;

  // 向命令队列添加一条指令来设置一个新的围栏点
  // 由于当前的GPU正在执行绘制命令，所以在GPU处理完Signal()函数之前的所有命令以
前，
  // 并不会设置此新的围栏点
  mCommandQueue->Signal(mFence.Get(), mCurrentFence);

  // 值得注意的是，GPU此时可能仍然在处理上一帧数据，但是这也没什么问题，因为我们
```

```
这些操作并没有
    // 影响与之前帧相关联的帧资源
}
```

不难看出，这种解决方案还是无法完全避免等待情况的发生。如果两种处理器处理帧的速度差距过大，则前者终将不得不等待后来者追上，因为差距过大将导致帧资源数据被错误地复写。如果GPU处理命令的速度快于CPU提交命令列表的速度，则GPU会进入空闲状态。通常来讲，若要尝试淋漓尽致地发挥系统图形处理方面的能力，就应当避免此情况的发生，因为这并没有充分利用GPU资源。此外，如果CPU处理帧的速度总是遥遥领先于GPU，则CPU一定存在等待的时间。而这正是我们所期待的情景，因为这使GPU被完全调动了起来，而CPU多出来的空闲时间总是可以被游戏的其他部分所利用，如AI（人工智能）、物理模拟以及游戏业务逻辑等。

因此，如果说采用多个帧资源也无法避免等待现象的发生，那么它对我们究竟有何用处呢？答案是：它使我们可以持续向GPU提供数据。也就是说，当GPU在处理第n帧的命令时，CPU可以继续构建和提交绘制第$n+1$帧和第$n+2$帧所用的命令。这将令命令队列保持非空状态，从而使GPU总有任务去执行。

7.2 渲染项

绘制一个物体需要设置多种参数，例如绑定顶点缓冲区和索引缓冲区、绑定与物体有关的常量数据、设定图元类型以及指定DrawIndexedInstanced方法的参数。随着场景中所绘物体的逐渐增多，如果我们能创建一个轻量级结构来存储绘制物体所需的数据，那真是极好的；由于每个物体的特征不同，绘制过程中所需的数据也会有所变化，因此该结构中的数据也会因具体程序而异。我们把单次绘制调用过程中，需要向渲染流水线提交的数据集称为渲染项（render item）。对于当前的演示程序而言，渲染项RenderItem结构体如下：

```
// 存储绘制图形所需参数的轻量级结构体。它会随着不同的应用程序而有所差别
struct RenderItem
{
  RenderItem() = default;

  // 描述物体局部空间相对于世界空间的世界矩阵
  // 它定义了物体位于世界空间中的位置、朝向以及大小
  XMFLOAT4X4 World = MathHelper::Identity4x4();

  // 用已更新标志（dirty flag）来表示物体的相关数据已发生改变，这意味着我们此时需要更新常量缓
  // 冲区。由于每个FrameResource中都有一个物体常量缓冲区，所以我们必须对每个FrameResource
  // 都进行更新。即，当我们修改物体数据的时候，应当按NumFramesDirty = gNumFrameResources
  // 进行设置，从而使每个帧资源都得到更新
  int NumFramesDirty = gNumFrameResources;

  // 该索引指向的GPU常量缓冲区对应于当前渲染项中的物体常量缓冲区
  UINT ObjCBIndex = -1;

  // 此渲染项参与绘制的几何体。注意，绘制一个几何体可能会用到多个渲染项
  MeshGeometry* Geo = nullptr;

  // 图元拓扑
```

```
  D3D12_PRIMITIVE_TOPOLOGY PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLE
LIST;

  // DrawIndexedInstanced方法的参数
  UINT IndexCount = 0;
  UINT StartIndexLocation = 0;
  int BaseVertexLocation = 0;
};
```

我们的应用程序将根据各渲染项的绘制目的，把它们保存在不同的向量里。即按照不同PSO（流水线状态对象）所需的渲染项，将它们划分到不同的向量之中。

```
// 存有所有渲染项的向量
std::vector<std::unique_ptr<RenderItem>> mAllRitems;

// 根据PSO来划分渲染项
std::vector<RenderItem*> mOpaqueRitems;
std::vector<RenderItem*> mTransparentRitems;
```

7.3　渲染过程中所用到的常量数据

从7.1节中可以看到，我们在自己实现的FrameResource类中引进了一个新的常量缓冲区：

```
std::unique_ptr<UploadBuffer<PassConstants>> PassCB = nullptr;
```

随着演示代码复杂度的不断增加，该缓冲区中存储的数据内容（例如观察位置、观察矩阵与投影矩阵以及与屏幕（渲染目标）分辨率等相关的信息）会根据特定的渲染过程（rendering pass）而确定下来。其中也包含了与游戏计时有关的信息，它们是着色器程序中要访问的极有用的数据。注意，我们的演示程序可能不会用到所有的常量数据，但是它们的存在却使工作变得更加方便，而且提供这些额外的数据也只需少量开销。例如，虽然我们现在无须知道渲染目标的尺寸，但当要实现某些后期处理效果之时，这个信息将会派上用场。

```
cbuffer cbPass : register(b1)
{
  float4x4 gView;
  float4x4 gInvView;
  float4x4 gProj;
  float4x4 gInvProj;
  float4x4 gViewProj;
  float4x4 gInvViewProj;
  float3 gEyePosW;
  float cbPerObjectPad1;
  float2 gRenderTargetSize;
  float2 gInvRenderTargetSize;
  float gNearZ;
  float gFarZ;
  float gTotalTime;
  float gDeltaTime;
};
```

此时，我们也已修改了物体常量缓冲区（即cbPerObject），使之仅存储一个与物体有关的常量。就目前的情况而言，为了绘制物体，与之唯一相关的常量就是它的世界矩阵：

```
cbuffer cbPerObject : register(b0)
{
    float4x4 gWorld;
};
```

我们做出上述调整的思路为：基于资源的更新频率对常量数据进行分组。在每次渲染过程（render pass）中，只需将本次所用的常量（cbPass）更新一次；而每当某个物体的世界矩阵发生改变时，只需更新该物体的相关常量（cbPerObject）即可。如果场景中有一个静态物体，比如一棵树，则只需对它的物体常量缓冲区设置一次（树的）世界矩阵，而后就再也不必对它进行更新了。在我们的演示程序中，将通过下列方法来更新渲染过程常量缓冲区以及物体常量缓冲区。在绘制每一帧画面时，这两个方法都将被Update函数调用一次。

```
void ShapesApp::UpdateObjectCBs(const GameTimer& gt)
{
  auto currObjectCB = mCurrFrameResource->ObjectCB.get();
  for(auto& e : mAllRitems)
  {
    // 只要常量发生了改变就得更新常量缓冲区内的数据。而且要对每个帧资源都进行更新
    if(e->NumFramesDirty > 0)
    {
      XMMATRIX world = XMLoadFloat4x4(&e->World);

      ObjectConstants objConstants;
      XMStoreFloat4x4(&objConstants.World, XMMatrixTranspose(world));

      currObjectCB->CopyData(e->ObjCBIndex, objConstants);

      // 还需要对下一个FrameResource进行更新
```

```
      e->NumFramesDirty--;
    }
  }
}

void ShapesApp::UpdateMainPassCB(const GameTimer& gt)
{
  XMMATRIX view = XMLoadFloat4x4(&mView);
  XMMATRIX proj = XMLoadFloat4x4(&mProj);

  XMMATRIX viewProj = XMMatrixMultiply(view, proj);
  XMMATRIX invView = XMMatrixInverse(&XMMatrixDeterminant(view), view);
  XMMATRIX invProj = XMMatrixInverse(&XMMatrixDeterminant(proj), proj);
  XMMATRIX invViewProj = XMMatrixInverse(&XMMatrixDeterminant(viewProj),
    viewProj);

  XMStoreFloat4x4(&mMainPassCB.View, XMMatrixTranspose(view));
  XMStoreFloat4x4(&mMainPassCB.InvView, XMMatrixTranspose(invView));
  XMStoreFloat4x4(&mMainPassCB.Proj, XMMatrixTranspose(proj));
  XMStoreFloat4x4(&mMainPassCB.InvProj, XMMatrixTranspose(invProj));
  XMStoreFloat4x4(&mMainPassCB.ViewProj, XMMatrixTranspose(viewProj));
  XMStoreFloat4x4(&mMainPassCB.InvViewProj, XMMatrixTranspose(invViewProj)
);
  mMainPassCB.EyePosW = mEyePos;
  mMainPassCB.RenderTargetSize = XMFLOAT2((float)mClientWidth, (float)
    mClientHeight);
  mMainPassCB.InvRenderTargetSize = XMFLOAT2(1.0f / mClientWidth, 1.0f
    / mClientHeight);
  mMainPassCB.NearZ = 1.0f;
  mMainPassCB.FarZ = 1000.0f;
  mMainPassCB.TotalTime = gt.TotalTime();
  mMainPassCB.DeltaTime = gt.DeltaTime();

  auto currPassCB = mCurrFrameResource->PassCB.get();
  currPassCB->CopyData(0, mMainPassCB);
}
```

随着这些常量缓冲区结构的改变，我们也要对顶点着色器进行相应的更新：

```
VertexOut VS(VertexIn vin)
{
  VertexOut vout;

  // 将顶点变换到齐次裁剪空间
```

```
  float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
  vout.PosH = mul(posW, gViewProj);

  // 直接向像素着色器传递顶点的颜色数据
  vout.Color = vin.Color;

  return vout;
}
```

此调整会引起每个顶点都要额外进行一次向量与矩阵的乘法运算，这对现代的GPU来说都是小菜一碟，因为它有着极其强大的计算能力。

现在，着色器所期望的输入资源已发生了改变，因此我们需要相应地调整根签名来使之获取所需的两个描述符表（此时，我们的着色器程序需要获取两个描述符表，因为这两个CBV（常量缓冲区视图）有着不同的更新频率——渲染过程CBV仅需在每个渲染过程中设置一次，而物体CBV则要针对每一个渲染项进行配置）：

```
CD3DX12_DESCRIPTOR_RANGE cbvTable0;
cbvTable0.Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 0);

CD3DX12_DESCRIPTOR_RANGE cbvTable1;
cbvTable1.Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 1);

// 根参数可能是描述符表、根描述符或根常量
CD3DX12_ROOT_PARAMETER slotRootParameter[2];

// 创建根CBV
slotRootParameter[0].InitAsDescriptorTable(1, &cbvTable0);
slotRootParameter[1].InitAsDescriptorTable(1, &cbvTable1);

// 根签名由一系列根参数所构成
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(2, slotRootParameter, 0, nullptr,
  D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
```

注意 Note

不要在着色器内使用过多的常量缓冲区。根据[Thibieroz13]提出的建议，出于性能的考虑，常量缓冲区的数量以少于5个为宜。

7.4　不同形状的几何体

在本节中，我们将展示如何创建不同形状的几何体，如椭球体、球体、柱体（通过调整圆柱体的上下两底即可创建出圆台和圆锥体）。这些几何体对于绘制天空穹顶（sky dome，即描述游戏中玩家头顶的天空部分，也有译作天空穹、天穹等）、图形程序调试、碰撞检测的可视化以及延迟渲染（deferred rendering）是有极大裨益的。比如说，我们可以先将正在制作中的游戏角色简化渲染成球体，以供调试检测。

我们将程序性几何体（procedural geometry，也有译作过程化几何体。这个词的译法较多，大意就是“根据用户提供的参数以程序自动生成对应的几何体”）的生成代码放入GeometryGenerator类（GeometryGenerator.h/.cpp）中。GeometryGenerator是一个工具类，用于生成如栅格、球体、柱体以及长方体这类简单的几何体，在我们的演示程序中将常常见到它们的身影。此类将数据生成在系统内存中，而我们必须将这些数据复制到顶点缓冲区和索引缓冲区内。GeometryGenerator类还可创建出一些后续章节要用到的顶点数据，由于我们当前的演示程序中还用不到它们，所以暂时不会将这些数据复制到顶点缓冲区。MeshData是一个嵌套在GeometryGenerator类中用于存储顶点列表和索引列表的简易结构体：

```
class GeometryGenerator
{
public:
  using uint16 = std::uint16_t;
  using uint32 = std::uint32_t;
```

```
struct Vertex
{
  Vertex(){}
  Vertex(
    const DirectX::XMFLOAT3& p,
    const DirectX::XMFLOAT3& n,
    const DirectX::XMFLOAT3& t,
    const DirectX::XMFLOAT2& uv) :
    Position(p),
    Normal(n),
    TangentU(t),
    TexC(uv){}
  Vertex(
    float px, float py, float pz,
    float nx, float ny, float nz,
    float tx, float ty, float tz,
    float u, float v) :
    Position(px,py,pz),
    Normal(nx,ny,nz),
    TangentU(tx, ty, tz),
    TexC(u,v){}

  DirectX::XMFLOAT3 Position;
  DirectX::XMFLOAT3 Normal;
  DirectX::XMFLOAT3 TangentU;
  DirectX::XMFLOAT2 TexC;
};

struct MeshData
{
  std::vector<Vertex> Vertices;
  std::vector<uint32> Indices32;

  std::vector<uint16>& GetIndices16()
  {
    if(mIndices16.empty())
    {
      mIndices16.resize(Indices32.size());
      for(size_t i = 0; i < Indices32.size(); ++i)
        mIndices16[i] = static_cast<uint16>(Indices32[i]);
    }

    return mIndices16;
  }

  private:
```

```
        std::vector<uint16> mIndices16;
    };

...

};
```

7.4.1　生成柱体网格

在定义一个柱体时，需要指定其顶、底面半径，高度，切片数量（slice count，即将截面分割的块数），以及堆叠层数（stack count，即横向切割的层数），如图7.1所示。程序中的柱体呈圆台形状，因而就此展开讨论。我们将圆台的构成分为侧面几何体，顶面几何体以及底面几何体3个部分。

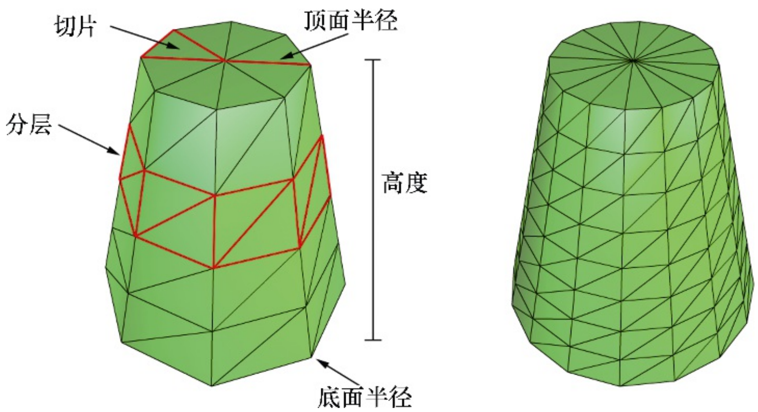


图7.1　左侧圆台被分为8个切片，划为4层。右侧圆台的被分为16个切片，划为8层。切片数量和堆叠层数控制着构成圆台的三角形密集程度，三角形越多则所绘图形越接近预定的几何体。注意，圆台顶面半径和底面半径是不同的，所以我们可以借此创建出趋向于锥体的几何体，而不仅仅是“正”圆柱体

7.4.1.1　柱体的侧面几何体

我们要生成的是中心（即1/2高度处截面的中心点）位于原点，且旋转轴平行于y轴的圆台。从图7.1中可以看出，圆台的所有顶点都列于其各层侧面的“环”上，共有stackCount + 1环，而每个环上的顶点数量都为sliceCount。相邻环的半径差为$\Delta r = (\text{topRadius-bottomRadius})/\text{stackCount}$。如果从底面上的环开始用索引来0表示，那么第$i$环的半径就是$r_i = bottomRadius + i\ \Delta r$，且第$i$环的高度值为$h_i = -\frac{h}{2} + i\Delta h$（可见，1/2高度以下为负值，1/2高度以上为正值），其中的Δh是每层的高度，h为圆台的高度。由此可知，生成圆台的基本思路是遍历每个环，并生成列于环上的各个顶点。下面给出此算法的实现：

```
GeometryGenerator::MeshData
GeometryGenerator::CreateCylinder(
  float bottomRadius, float topRadius,
  float height, uint32 sliceCount, uint32 stackCount)
{
  MeshData meshData;

  //
  // 构建堆叠层
  //

  float stackHeight = height / stackCount;

  // 计算从下至上遍历每个相邻分层时所需的半径增量
  float radiusStep = (topRadius - bottomRadius) / stackCount;

  uint32 ringCount = stackCount+1;

  // 从底面开始，由下至上计算每个堆叠层环上的顶点坐标
  for(uint32 i = 0; i < ringCount; ++i)
  {
    float y = -0.5f*height + i*stackHeight;
    float r = bottomRadius + i*radiusStep;

    // 环上的各个顶点
```

```
    float dTheta = 2.0f*XM_PI/sliceCount;
    for(uint32 j = 0; j <= sliceCount; ++j)
    {
      Vertex vertex;

      float c = cosf(j*dTheta);
      float s = sinf(j*dTheta);

      vertex.Position = XMFLOAT3(r*c, y, r*s);

      vertex.TexC.x = (float)j/sliceCount;
      vertex.TexC.y = 1.0f - (float)i/stackCount;

      // 可以像下面那样以参数化（parameterized）的方式来计算圆台顶点，我们引入
与纹理坐标v方
      // 向相同的参数v，从而使副切线（bitangent，相关概念见19.3节）与纹理坐标v
      // 的方向相同
      // 设r0为底面半径，r1为顶面半径
      //  y(v) = h - hv  其中v位于区间[0,1]
      //  r(v) = r1 + (r0-r1)v
      //
      //  x(t, v) = r(v)*cos(t)
      //  y(t, v) = h - hv
      //  z(t, v) = r(v)*sin(t)
      //
      // dx/dt = -r(v)*sin(t)
      // dy/dt = 0
      // dz/dt = +r(v)*cos(t)
      //
      // dx/dv = (r0-r1)*cos(t)
      // dy/dv = -h
      // dz/dv = (r0-r1)*sin(t)

      // 此为单位长度
      vertex.TangentU = XMFLOAT3(-s, 0.0f, c);

      float dr = bottomRadius-topRadius;
      XMFLOAT3 bitangent(dr*c, -height, dr*s);

      XMVECTOR T = XMLoadFloat3(&vertex.TangentU);
      XMVECTOR B = XMLoadFloat3(&bitangent);
      XMVECTOR N = XMVector3Normalize(XMVector3Cross(T, B));
      XMStoreFloat3(&vertex.Normal, N);

      meshData.Vertices.push_back(vertex);
    }
  }
```

注意 Note

从上述代码中可以看出，每个环上的第一个顶点与最后一个顶点在位置上是重合的，但是二者的纹理坐标却并不相同。只有这样做才能保证在圆台上绘制出正确的纹理。

注意 Note

由`GeometryGenerator::CreateCylinder`方法创建的其他顶点数据（如法向量和纹理坐标）对后续章节中的演示程序而言是不可或缺的，但我们现在还不会用到它们。

观察图7.2可知，由每个分层以及切片分割出的侧面块都是一个四边形（由两个三角形构成）。而以第i层与第j块切片所确定下来的侧面块中的两个三角形的索引分别为：

$$\Delta ABC = (i \cdot n + j, (i+1) \cdot n + j, (i+1) \cdot n + j + 1)$$
$$\Delta ACD = (i \cdot n + j, (i+1) \cdot n + j + 1, i \cdot n + j + 1)$$

其中，n是每个环上的顶点数量。因此，求取圆台侧面块上所有三角形索引的主要思路是：遍历每个堆叠层和每个切片，并运用上述公式

进行计算。

```
  // +1是希望让每环的第一个顶点和最后一个顶点重合，这是因为它们的纹理坐标并不相同
  uint32 ringVertexCount = sliceCount+1;

  // 计算每个侧面块中三角形的索引
  for(uint32 i = 0; i < stackCount; ++i)
  {
    for(uint32 j = 0; j < sliceCount; ++j)
    {
      meshData.Indices32.push_back(i*ringVertexCount + j);
      meshData.Indices32.push_back((i+1)*ringVertexCount + j);
      meshData.Indices32.push_back((i+1)*ringVertexCount + j+1);

      meshData.Indices32.push_back(i*ringVertexCount + j);
      meshData.Indices32.push_back((i+1)*ringVertexCount + j+1);
      meshData.Indices32.push_back(i*ringVertexCount + j+1);
    }
  }

  BuildCylinderTopCap(bottomRadius, topRadius, height,
    sliceCount, stackCount, meshData);
  BuildCylinderBottomCap(bottomRadius, topRadius, height,
    sliceCount, stackCount, meshData);

  return meshData;
}
```

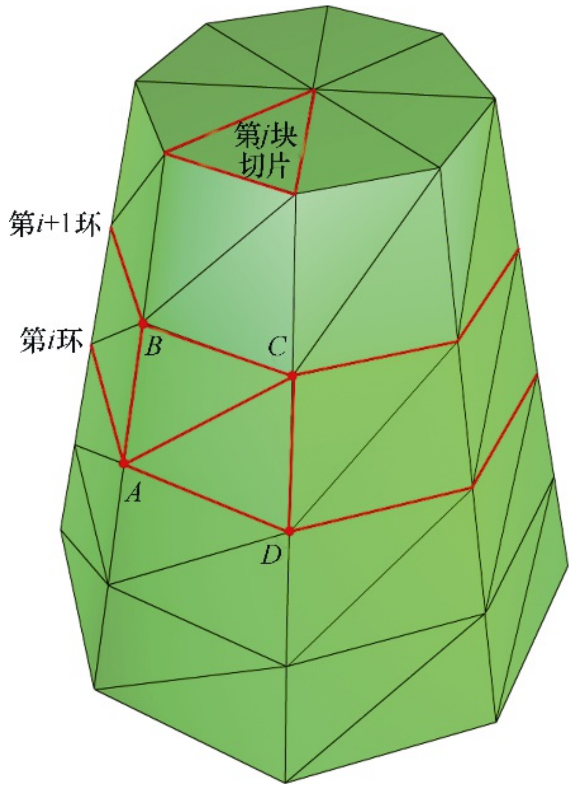


图7.2　顶点A、B、C、D分别位于第i环、第$i+1$环以及第j块切片所合围的侧面块之中

7.4.1.2　柱体的端面几何体

生成圆台端面的几何体，相当于在其顶面和底面的截面上切割出多个三角形，使之逼近一个圆形：

```
void GeometryGenerator::BuildCylinderTopCap(
  float bottomRadius, float topRadius, float height,
  uint32 sliceCount, uint32 stackCount, MeshData& meshData)
{
  uint32 baseIndex = (uint32)meshData.Vertices.size();

  float y = 0.5f*height;
  float dTheta = 2.0f*XM_PI/sliceCount;

  // 使圆台端面环上的首尾顶点重合，因为这两个顶点的纹理坐标和法线是不同的
  for(uint32 i = 0; i <= sliceCount; ++i)
  {
    float x = topRadius*cosf(i*dTheta);
```

```
    float z = topRadius*sinf(i*dTheta);

    // 根据圆台的高度使顶面纹理坐标的范围按比例缩小
    float u = x/height + 0.5f;
    float v = z/height + 0.5f;

    meshData.Vertices.push_back(
      Vertex(x, y, z, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, u, v) );
  }

  // 顶面的中心顶点
  meshData.Vertices.push_back(
    Vertex(0.0f, y, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.5f, 0.5f)
);

  // 中心顶点的索引值
  uint32 centerIndex = (uint32)meshData.Vertices.size()-1;

  for(uint32 i = 0; i < sliceCount; ++i)
  {
    meshData.Indices32.push_back(centerIndex);
    meshData.Indices32.push_back(baseIndex + i+1);
    meshData.Indices32.push_back(baseIndex + i);
  }
}
```

生成圆台底面的代码与之相似。

7.4.2　生成球体网格

欲定义一个球体，就要指定其半径、切片数量及其堆叠层数，如图7.3所示。除了每个环上的半径是依三角函数非线性变化，生成球体的算法与生成圆台的算法非常相近。我们将把GeometryGenerator::CreateSphere方法的代码留给读者自行研究。最后，值得一提的是，若采用不等比缩放世界变换，即可将球体转换为椭球体。

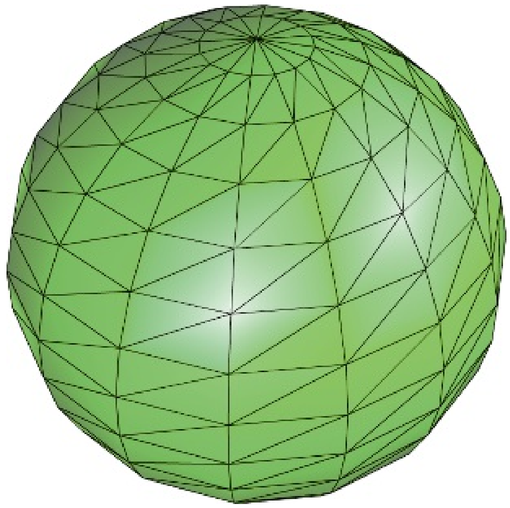


图7.3　柱体的切片与分层思想也同样可以应用到球体上，借此来控制球体的曲面细分级别

7.4.3　生成几何球体网格

观察图7.3可知，构成球体的三角形面积并不相同，这在某些情景中并非我们所愿。相对而言，几何球体（geosphere）利用面积相同且边长相等的三角形来逼近球体，如图7.4所示。

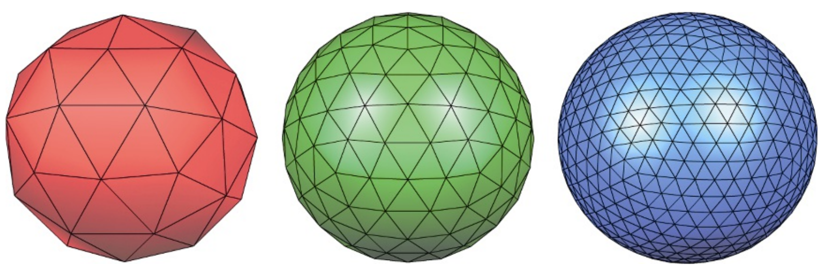


图7.4　通过反复细分并将新生成的顶点重新投影到球面上，便可以近似地表示一个几何球体

为了生成几何球体，我们以一个正二十面体作为基础，细分其上的三角形，再根据给定的半径向球面投影新生成的顶点。反复重复这个过程，便可以提高该几何球体的曲面细分程度。

图7.5展示了如何将一个三角形细分为4个大小相等的小三角形。不难发现，新生成的顶点都位于原始三角形边上的中点。先将顶点投影到单位球面上，再利用r进行标量乘法：$\boldsymbol{v}' = r\frac{\boldsymbol{v}}{\|\boldsymbol{v}\|}$，即可把新顶点都投影到半径为$r$的球体之上。

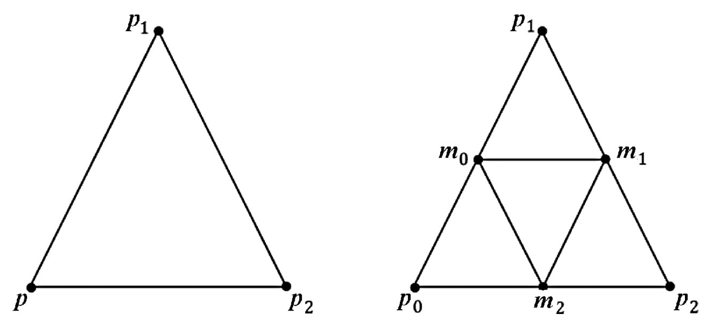


图7.5　将一个三角形细分为4个等面积小三角形的过程

相应的代码如下：

```
GeometryGenerator::MeshData
GeometryGenerator::CreateGeosphere(float radius, uint32 numSubdivisions)
{
  MeshData meshData;

  // 确定细分的次数
  numSubdivisions = std::min<uint32>(numSubdivisions, 6u);

  // 通过对一个正二十面体进行曲面细分来逼近一个球体

  const float X = 0.525731f;
  const float Z = 0.850651f;

  XMFLOAT3 pos[12] =
  {
    XMFLOAT3(-X, 0.0f, Z),  XMFLOAT3(X, 0.0f, Z),
    XMFLOAT3(-X, 0.0f, -Z), XMFLOAT3(X, 0.0f, -Z),
    XMFLOAT3(0.0f, Z, X),   XMFLOAT3(0.0f, Z, -X),
    XMFLOAT3(0.0f, -Z, X),  XMFLOAT3(0.0f, -Z, -X),
    XMFLOAT3(Z, X, 0.0f),   XMFLOAT3(-Z, X, 0.0f),
    XMFLOAT3(Z, -X, 0.0f),  XMFLOAT3(-Z, -X, 0.0f)
  };
```

```
uint32 k[60] =
{
  1,4,0,  4,9,0,  4,5,9,  8,5,4,  1,8,4,
  1,10,8, 10,3,8, 8,3,5,  3,2,5,  3,7,2,
  3,10,7, 10,6,7, 6,11,7, 6,0,11, 6,1,0,
  10,1,6, 11,0,9, 2,11,9, 5,2,9,  11,2,7
};

meshData.Vertices.resize(12);
meshData.Indices32.assign(&k[0], &k[60]);

for(uint32 i = 0; i < 12; ++i)
  meshData.Vertices[i].Position = pos[i];

for(uint32 i = 0; i < numSubdivisions; ++i)
  Subdivide(meshData);

// 将每一个顶点都投影到球面，并推导其对应的纹理坐标
for(uint32 i = 0; i < meshData.Vertices.size(); ++i)
{
  // 投影到单位球面上
  XMVECTOR n = XMVector3Normalize(XMLoadFloat3(&meshData.Vertices[i].
  Position));

  // 投射到球面上
  XMVECTOR p = radius*n;

  XMStoreFloat3(&meshData.Vertices[i].Position, p);
  XMStoreFloat3(&meshData.Vertices[i].Normal, n);

  // 根据球面坐标推导出纹理坐标
  float theta = atan2f(meshData.Vertices[i].Position.z,
             meshData.Vertices[i].Position.x);

  // 将theta限制在[0, 2pi]区间内
  if(theta < 0.0f)
    theta += XM_2PI;

  float phi = acosf(meshData.Vertices[i].Position.y / radius);

  meshData.Vertices[i].TexC.x = theta/XM_2PI;
  meshData.Vertices[i].TexC.y = phi/XM_PI;

  // 求出P关于theta（p对于theta）的偏导数
  meshData.Vertices[i].TangentU.x = -radius*sinf(phi)*sinf(theta);
  meshData.Vertices[i].TangentU.y = 0.0f;
  meshData.Vertices[i].TangentU.z = +radius*sinf(phi)*cosf(theta);
```

```
    XMVECTOR T = XMLoadFloat3(&meshData.Vertices[i].TangentU);
    XMStoreFloat3(&meshData.Vertices[i].TangentU, XMVector3Normalize(T));
  }

  return meshData;
}
```

7.5　绘制多种几何体演示程序

为了示范生成球体和柱体的代码，我们实现了效果如图7.6所示的Shapes演示程序。另外，在探究该例程的过程中，我们将获得在场景中（利用创建多个世界变换矩阵）绘制多个物体并对其进行定位的宝贵经验。此外，我们会将场景中的所有几何体都置于一个大的顶点缓冲区和一个索引缓冲区中。最后，通过多次调用`DrawIndexedInstanced`方法来绘制每一个物体（即每次调用仅绘制一个物体，因为不同物体的世界矩阵也各不相同）。因此，我们将看到一个涉及修改`DrawIndexedInstanced`方法中`StartIndexLocation`和`BaseVertexLocation`参数的调用示例。

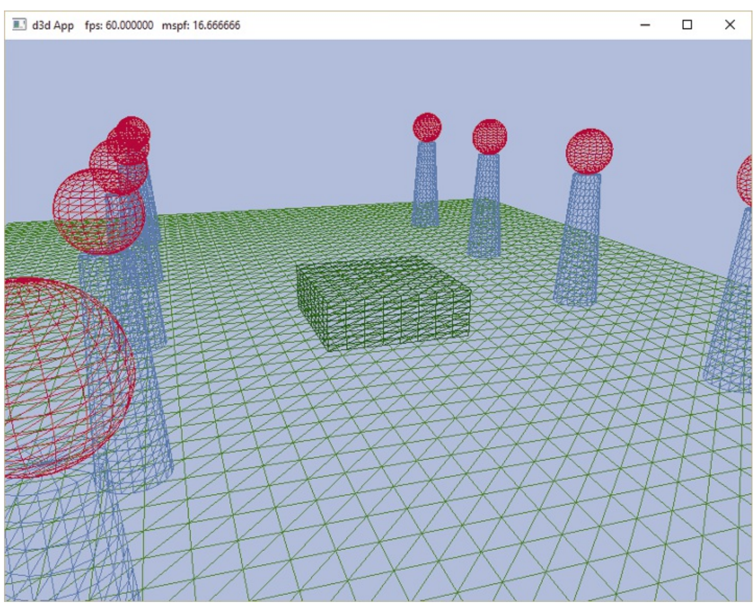


图7.6　Shapes演示程序的效果

7.5.1　顶点缓冲区和索引缓冲区

正如图7.6所示，在Shapes演示程序中，我们要绘制长方体、栅格、柱体（圆台）及球体。尽管在示例中要绘制多个球体和柱体，但实际上我们只需对一组球体和圆台数据的副本。通过利用不同的世界矩阵，对这组数据进行多次绘制，即可绘制出多个预定的球体与圆台。所以说，这也是一个几何体实例化（instancing）的范例，借助此技术可以减少内存资源的占用。

通过将不同物体的顶点和索引合并起来，我们把所有几何体网格的顶点和索引都装进一个顶点缓冲区及一个索引缓冲区内。这意味着，在绘制一个物体时，我们只需绘制此物体位于顶点和索引这两种缓冲区中的数据子集。为了调用`ID3D12GraphicsCommandList::DrawIndexedInstanced`方法而仅绘制几何体的子集，我们需要掌握3种数据（可参照图6.3并回顾6.3节的相关讨论），它们分别是：待绘制物体在合并索引缓冲区中的起始索引、待绘制物体的索引数量，以及基准顶点地址——即待绘制物体第一个顶点在合并顶点缓冲区中的索引。经回顾可知，基准顶点地址是一个整数值，我们要在绘制调用过程中、获取顶点数据之前，将它与物体的原索引值依次相加，以此获取合并顶点缓冲区中所绘物体的正确顶点子集（参见第5章的练习2）。

下列代码展示了创建几何体缓冲区、缓存绘制调用所需参数值以及绘制物体的具体过程。

```
void ShapesApp::BuildShapeGeometry()
{
```

```
  GeometryGenerator geoGen;
  GeometryGenerator::MeshData box = geoGen.CreateBox(1.5f, 0.5f, 1.5f, 3);
  GeometryGenerator::MeshData grid = geoGen.CreateGrid(20.0f, 30.0f, 60, 4
0);
  GeometryGenerator::MeshData sphere = geoGen.CreateSphere(0.5f, 20, 20);
  GeometryGenerator::MeshData cylinder = geoGen.CreateCylinder(0.5f,
    0.3f, 3.0f, 20, 20);

  //
  // 将所有的几何体数据都合并到一对大的顶点/索引缓冲区中
  // 以此来定义每个子网格数据在缓冲区中所占的范围
  //

  // 对合并顶点缓冲区中每个物体的顶点偏移量进行缓存
  UINT boxVertexOffset = 0;
  UINT gridVertexOffset = (UINT)box.Vertices.size();
  UINT sphereVertexOffset = gridVertexOffset + (UINT)grid.Vertices.size();
  UINT cylinderVertexOffset = sphereVertexOffset + (UINT)sphere.Vertices.s
ize();

  // 对合并索引缓冲区中每个物体的起始索引进行缓存
  UINT boxIndexOffset = 0;
  UINT gridIndexOffset = (UINT)box.Indices32.size();
  UINT sphereIndexOffset = gridIndexOffset + (UINT)grid.Indices32.size();
  UINT cylinderIndexOffset = sphereIndexOffset + (UINT)sphere.Indices32.si
ze();

  // 定义的多个SubmeshGeometry结构体中包含了顶点/索引缓冲区内不同几何体的子网格
数据

  SubmeshGeometry boxSubmesh;
  boxSubmesh.IndexCount = (UINT)box.Indices32.size();
  boxSubmesh.StartIndexLocation = boxIndexOffset;
  boxSubmesh.BaseVertexLocation = boxVertexOffset;

  SubmeshGeometry gridSubmesh;
  gridSubmesh.IndexCount = (UINT)grid.Indices32.size();
  gridSubmesh.StartIndexLocation = gridIndexOffset;
  gridSubmesh.BaseVertexLocation = gridVertexOffset;

  SubmeshGeometry sphereSubmesh;
  sphereSubmesh.IndexCount = (UINT)sphere.Indices32.size();
  sphereSubmesh.StartIndexLocation = sphereIndexOffset;
  sphereSubmesh.BaseVertexLocation = sphereVertexOffset;

  SubmeshGeometry cylinderSubmesh;
  cylinderSubmesh.IndexCount = (UINT)cylinder.Indices32.size();
  cylinderSubmesh.StartIndexLocation = cylinderIndexOffset;
```

```
cylinderSubmesh.BaseVertexLocation = cylinderVertexOffset;

//
// 提取出所需的顶点元素，再将所有网格的顶点装进一个顶点缓冲区
//

auto totalVertexCount =
  box.Vertices.size() +
  grid.Vertices.size() +
  sphere.Vertices.size() +
  cylinder.Vertices.size();

std::vector<Vertex> vertices(totalVertexCount);

UINT k = 0;
for(size_t i = 0; i < box.Vertices.size(); ++i, ++k)
{
  vertices[k].Pos = box.Vertices[i].Position;
  vertices[k].Color = XMFLOAT4(DirectX::Colors::DarkGreen);
}

for(size_t i = 0; i < grid.Vertices.size(); ++i, ++k)
{
  vertices[k].Pos = grid.Vertices[i].Position;
  vertices[k].Color = XMFLOAT4(DirectX::Colors::ForestGreen);
}

for(size_t i = 0; i < sphere.Vertices.size(); ++i, ++k)
{
  vertices[k].Pos = sphere.Vertices[i].Position;
  vertices[k].Color = XMFLOAT4(DirectX::Colors::Crimson);
}

for(size_t i = 0; i < cylinder.Vertices.size(); ++i, ++k)
{
  vertices[k].Pos = cylinder.Vertices[i].Position;
  vertices[k].Color = XMFLOAT4(DirectX::Colors::SteelBlue);
}

std::vector<std::uint16_t> indices;
indices.insert(indices.end(),
  std::begin(box.GetIndices16()),
  std::end(box.GetIndices16()));
indices.insert(indices.end(),
  std::begin(grid.GetIndices16()),
  std::end(grid.GetIndices16()));
indices.insert(indices.end(),
```

```
    std::begin(sphere.GetIndices16()),
    std::end(sphere.GetIndices16()));
  indices.insert(indices.end(),
    std::begin(cylinder.GetIndices16()),
    std::end(cylinder.GetIndices16()));

  const UINT vbByteSize = (UINT)vertices.size() * sizeof(Vertex);
  const UINT ibByteSize = (UINT)indices.size() * sizeof(std::uint16_t);

  auto geo = std::make_unique<MeshGeometry>();
  geo->Name = "shapeGeo";

  ThrowIfFailed(D3DCreateBlob(vbByteSize, &geo->VertexBufferCPU));
  CopyMemory(geo->VertexBufferCPU->GetBufferPointer(), vertices.data(),
  vbByteSize);

  ThrowIfFailed(D3DCreateBlob(ibByteSize, &geo->IndexBufferCPU));
  CopyMemory(geo->IndexBufferCPU->GetBufferPointer(), indices.data(),
  ibByteSize);

  geo->VertexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
    mCommandList.Get(), vertices.data(), vbByteSize, geo-
    >VertexBufferUploader);

  geo->IndexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
    mCommandList.Get(), indices.data(), ibByteSize, geo-
    >IndexBufferUploader);

  geo->VertexByteStride = sizeof(Vertex);
  geo->VertexBufferByteSize = vbByteSize;
  geo->IndexFormat = DXGI_FORMAT_R16_UINT;
  geo->IndexBufferByteSize = ibByteSize;

  geo->DrawArgs["box"] = boxSubmesh;
  geo->DrawArgs["grid"] = gridSubmesh;
  geo->DrawArgs["sphere"] = sphereSubmesh;
  geo->DrawArgs["cylinder"] = cylinderSubmesh;

  mGeometries[geo->Name] = std::move(geo);
}
```

上述方法中最后一行所用到的变量mGeometries被定义为：

```
std::unordered_map<std::string, std::unique_ptr<MeshGeometry>> mGeometries;
```

这是我们在本书后面常会用到的一种通用模式：为每个几何体、PSO、纹理和着色器等创建新的变量名是一件很烦人的事，所以我们使用无序映射表（unordered map），并根据名称在常数时间（constant time）内寻找和引用所需的对象。以下是一些相关的范例：

```
std::unordered_map<std::string, std::unique_ptr<MeshGeometry>> mGeometries
;
std::unordered_map<std::string, ComPtr<ID3DBlob>> mShaders;
std::unordered_map<std::string, ComPtr<ID3D12PipelineState>> mPSOs;
```

7.5.2 渲染项

现在我们来定义场景中的渲染项。通过观察以下代码可知，所有的渲染项共用同一个`MeshGeometry`，所以，我们通过`DrawArgs`获取`DrawIndexedInstanced`方法的参数，并以此来绘制顶点/索引缓冲区的子区域（也就是单个几何体）。

```
// ShapesApp类中的成员变量
std::vector<std::unique_ptr<RenderItem>> mAllRitems;
std::vector<RenderItem*> mOpaqueRitems;

void ShapesApp::BuildRenderItems()
{
  auto boxRitem = std::make_unique<RenderItem>();
  XMStoreFloat4x4(&boxRitem->World,
    XMMatrixScaling(2.0f, 2.0f, 2.0f)*XMMatrixTranslation(0.0f, 0.5f, 0.0f
));
  boxRitem->ObjCBIndex = 0;
  boxRitem->Geo = mGeometries["shapeGeo"].get();
  boxRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST;
  boxRitem->IndexCount = boxRitem->Geo->DrawArgs["box"].IndexCount;
  boxRitem->StartIndexLocation = boxRitem->Geo->DrawArgs["box"].
   StartIndexLocation;
  boxRitem->BaseVertexLocation = boxRitem->Geo->DrawArgs["box"].
   BaseVertexLocation;
```

```
  mAllRitems.push_back(std::move(boxRitem));

  auto gridRitem = std::make_unique<RenderItem>();
  gridRitem->World = MathHelper::Identity4x4();
  gridRitem->ObjCBIndex = 1;
  gridRitem->Geo = mGeometries["shapeGeo"].get();
  gridRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST;
  gridRitem->IndexCount = gridRitem->Geo->DrawArgs["grid"].IndexCount;
  gridRitem->StartIndexLocation = gridRitem->Geo->DrawArgs["grid"].
   StartIndexLocation;
  gridRitem->BaseVertexLocation = gridRitem->Geo->DrawArgs["grid"].
   BaseVertexLocation;
  mAllRitems.push_back(std::move(gridRitem));

  // 构建图7.6所示的柱体和球体
  UINT objCBIndex = 2;
  for(int i = 0; i < 5; ++i)
  {
    auto leftCylRitem = std::make_unique<RenderItem>();
    auto rightCylRitem = std::make_unique<RenderItem>();
    auto leftSphereRitem = std::make_unique<RenderItem>();
    auto rightSphereRitem = std::make_unique<RenderItem>();

    XMMATRIX leftCylWorld = XMMatrixTranslation(-5.0f, 1.5f, -10.0f + i*5.
0f);
    XMMATRIX rightCylWorld = XMMatrixTranslation(+5.0f, 1.5f, -10.0f + i*5
.0f);

    XMMATRIX leftSphereWorld = XMMatrixTranslation(-5.0f, 3.5f, -10.0f + i
*5.0f);
    XMMATRIX rightSphereWorld = XMMatrixTranslation(+5.0f, 3.5f, -10.0f +
i*5.0f);

    XMStoreFloat4x4(&leftCylRitem->World, rightCylWorld);
    leftCylRitem->ObjCBIndex = objCBIndex++;
    leftCylRitem->Geo = mGeometries["shapeGeo"].get();
    leftCylRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST;
    leftCylRitem->IndexCount = leftCylRitem->Geo->DrawArgs["cylinder"].
      IndexCount;
    leftCylRitem->StartIndexLocation =
      leftCylRitem->Geo->DrawArgs["cylinder"].StartIndexLocation;
    leftCylRitem->BaseVertexLocation =
      leftCylRitem->Geo->DrawArgs["cylinder"].BaseVertexLocation;

    XMStoreFloat4x4(&rightCylRitem->World, leftCylWorld);
    rightCylRitem->ObjCBIndex = objCBIndex++;
    rightCylRitem->Geo = mGeometries["shapeGeo"].get();
    rightCylRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST;
```

```
    rightCylRitem->IndexCount = rightCylRitem->Geo->DrawArgs["cylinder"].
      IndexCount;
    rightCylRitem->StartIndexLocation =
      rightCylRitem->Geo->DrawArgs["cylinder"].StartIndexLocation;
    rightCylRitem->BaseVertexLocation =
      rightCylRitem->Geo->DrawArgs["cylinder"].BaseVertexLocation;

    XMStoreFloat4x4(&leftSphereRitem->World, leftSphereWorld);
    leftSphereRitem->ObjCBIndex = objCBIndex++;
    leftSphereRitem->Geo = mGeometries["shapeGeo"].get();
    leftSphereRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST;
    leftSphereRitem->IndexCount = leftSphereRitem->Geo->DrawArgs["sphere"]
.
      IndexCount;
    leftSphereRitem->StartIndexLocation =
      leftSphereRitem->Geo->DrawArgs["sphere"].StartIndexLocation;
    leftSphereRitem->BaseVertexLocation =
      leftSphereRitem->Geo->DrawArgs["sphere"].BaseVertexLocation;

    XMStoreFloat4x4(&rightSphereRitem->World, rightSphereWorld);
    rightSphereRitem->ObjCBIndex = objCBIndex++;
    rightSphereRitem->Geo = mGeometries["shapeGeo"].get();
    rightSphereRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST;
    rightSphereRitem->IndexCount = rightSphereRitem->Geo->DrawArgs["sphere
"].
      IndexCount;
    rightSphereRitem->StartIndexLocation =
      rightSphereRitem->Geo->DrawArgs["sphere"].StartIndexLocation;
    rightSphereRitem->BaseVertexLocation =
      rightSphereRitem->Geo->DrawArgs["sphere"].BaseVertexLocation;

    mAllRitems.push_back(std::move(leftCylRitem));
    mAllRitems.push_back(std::move(rightCylRitem));
    mAllRitems.push_back(std::move(leftSphereRitem));
    mAllRitems.push_back(std::move(rightSphereRitem));
  }

  // 此演示程序中的所有渲染项都是非透明的
  for(auto& e : mAllRitems)
    mOpaqueRitems.push_back(e.get());
}
```

7.5.3　帧资源和常量缓冲区视图

回顾前文可知，我们已经创建了一个由FrameResource类型元素所构成的向量，每个FrameResource中都有上传缓冲区，用于为场景中的每个渲染项存储渲染过程常量以及物体常量数据。

```
std::unique_ptr<UploadBuffer<PassConstants>> PassCB = nullptr;
std::unique_ptr<UploadBuffer<ObjectConstants>> ObjectCB = nullptr;
```

如果有3个帧资源与n个渲染项，那么就应存在$3n$个物体常量缓冲区（object constant buffer）以及3个渲染过程常量缓冲区（pass constant buffer）。因此，我们也就需要创建$3(n+1)$个常量缓冲区视图（CBV）。这样一来，我们还要修改CBV堆以容纳额外的描述符：

```
void ShapesApp::BuildDescriptorHeaps()
{
  UINT objCount = (UINT)mOpaqueRitems.size();

  // 我们需要为每个帧资源中的每一个物体都创建一个CBV描述符，
  // 为了容纳每个帧资源中的渲染过程CBV而+1
  UINT numDescriptors = (objCount+1) * gNumFrameResources;

  // 保存渲染过程CBV的起始偏移量。在本程序中，这是排在最后面的3个描述符
  mPassCbvOffset = objCount * gNumFrameResources;

  D3D12_DESCRIPTOR_HEAP_DESC cbvHeapDesc;
  cbvHeapDesc.NumDescriptors = numDescriptors;
  cbvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
  cbvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
  cbvHeapDesc.NodeMask = 0;
  ThrowIfFailed(md3dDevice->CreateDescriptorHeap(&cbvHeapDesc,
    IID_PPV_ARGS(&mCbvHeap)));
}
```

现在，我们就可以用下列代码来填充CBV堆，其中描述符0至描述符$n-1$包含了第0个帧资源的物体CBV，描述符n至描述符$2n-1$容纳了第1个帧资源的物体CBV，以此类推，描述符$2n$至描述符$3n-1$包含了第2个

帧资源的物体CBV。最后，$3n$、$3n+1$以及$3n+2$分别存有第0个、第1个和第2个帧资源的渲染过程CBV：

```
void ShapesApp::BuildConstantBufferViews()
{
  UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof
   (ObjectConstants));

  UINT objCount = (UINT)mOpaqueRitems.size();

  // 每个帧资源中的每一个物体都需要一个对应的CBV描述符
  for(int frameIndex = 0; frameIndex < gNumFrameResources; ++frameIndex)
  {
    auto objectCB = mFrameResources[frameIndex]->ObjectCB->Resource();
    for(UINT i = 0; i < objCount; ++i)
    {
      D3D12_GPU_VIRTUAL_ADDRESS cbAddress = objectCB->GetGPUVirtualAddress
();

      // 偏移到缓冲区中第i个物体的常量缓冲区
      cbAddress += i*objCBByteSize;

      // 偏移到该物体在描述符堆中的CBV
      int heapIndex = frameIndex*objCount + i;
      auto handle = CD3DX12_CPU_DESCRIPTOR_HANDLE(
        mCbvHeap->GetCPUDescriptorHandleForHeapStart());
      handle.Offset(heapIndex, mCbvSrvUavDescriptorSize);

      D3D12_CONSTANT_BUFFER_VIEW_DESC cbvDesc;
      cbvDesc.BufferLocation = cbAddress;
      cbvDesc.SizeInBytes = objCBByteSize;

      md3dDevice->CreateConstantBufferView(&cbvDesc, handle);
    }
  }

  UINT passCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof
    (PassConstants));

  // 最后3个描述符依次是每个帧资源的渲染过程CBV
  for(int frameIndex = 0; frameIndex < gNumFrameResources; ++frameIndex)
  {
    auto passCB = mFrameResources[frameIndex]->PassCB->Resource();

    // 每个帧资源的渲染过程缓冲区中只存有一个常量缓冲区
```

```
    D3D12_GPU_VIRTUAL_ADDRESS cbAddress = passCB->GetGPUVirtualAddress();

    // 偏移到描述符堆中对应的渲染过程CBV
    int heapIndex = mPassCbvOffset + frameIndex;
    auto handle = CD3DX12_CPU_DESCRIPTOR_HANDLE(
      mCbvHeap->GetCPUDescriptorHandleForHeapStart());
    handle.Offset(heapIndex, mCbvSrvUavDescriptorSize);

    D3D12_CONSTANT_BUFFER_VIEW_DESC cbvDesc;
    cbvDesc.BufferLocation = cbAddress;
    cbvDesc.SizeInBytes = passCBByteSize;

    md3dDevice->CreateConstantBufferView(&cbvDesc, handle);
  }
}
```

前文已经讲过，通过调用ID3D12DescriptorHeap::GetCPUDescriptorHandleForHeapStar方法，我们可以获得堆中第一个描述符的句柄。然而，我们当前堆内所存放的描述符已不止一个，所以仅使用此方法并不能找到其他描述符的句柄。此时，我们希望能够偏移到堆内的其他描述符处，为此需要了解到达堆内下一个相邻描述符的增量。这个增量的大小其实是由硬件来确定的，所以我们必须从设备上查询相关的信息。此外，该增量还依赖于堆的具体类型。现在来重温在D3DApp类中缓存的下列信息：

```
mRtvDescriptorSize = md3dDevice->GetDescriptorHandleIncrementSize(
  D3D12_DESCRIPTOR_HEAP_TYPE_RTV);

mDsvDescriptorSize = md3dDevice->GetDescriptorHandleIncrementSize(
  D3D12_DESCRIPTOR_HEAP_TYPE_DSV);

mCbvSrvUavDescriptorSize = md3dDevice->GetDescriptorHandleIncrementSize(
  D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
```

只要知道了相邻描述符之间的增量大小，就能通过两种CD3DX12_CPU_DESCRIPTOR_HANDLE:: Offset方法之一偏移到第n个

描述符的句柄处：

```
// 指定要偏移到的目标描述符的编号，将它与相邻描述符之间的增量相乘，以此来找到第n
个描述符的句柄
CD3DX12_CPU_DESCRIPTOR_HANDLE handle = mCbvHeap->GetCPUDescriptorHandleFor
HeapStart();
handle.Offset(n * mCbvSrvDescriptorSize);

// 或者用另一个等价实现，先指定要偏移到第几个描述符，再给出描述符的增量大小
CD3DX12_CPU_DESCRIPTOR_HANDLE handle = mCbvHeap->GetCPUDescriptorHandleFor
HeapStart();
handle.Offset(n, mCbvSrvDescriptorSize);
```

注意 Note

CD3DX12_GPU_DESCRIPTOR_HANDLE有着同样的Offset方法。

7.5.4 绘制场景

最后一步就是绘制渲染项了，其中最复杂的部分莫过于为了绘制物体而根据偏移量找到它在堆中所对应的CBV。先来留心观察一下，渲染项是如何来存储一个索引，并使之指向与它有关的常量缓冲区。

```
void ShapesApp::DrawRenderItems(
  ID3D12GraphicsCommandList* cmdList,
  const std::vector<RenderItem*>& ritems)
{
  UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(
    ObjectConstants));
  auto objectCB = mCurrFrameResource->ObjectCB->Resource();
```

```
  // 对于每个渲染项来说...
  for(size_t i = 0; i < ritems.size(); ++i)
  {
    auto ri = ritems[i];

    cmdList->IASetVertexBuffers(0, 1, &ri->Geo->VertexBufferView());
    cmdList->IASetIndexBuffer(&ri->Geo->IndexBufferView());
    cmdList->IASetPrimitiveTopology(ri->PrimitiveType);

    // 为了绘制当前的帧资源和当前物体，偏移到描述符堆中对应的CBV处
    UINT cbvIndex = mCurrFrameResourceIndex*(UINT)mOpaqueRitems.size()
      + ri->ObjCBIndex;
    auto cbvHandle = CD3DX12_GPU_DESCRIPTOR_HANDLE(
      mCbvHeap->GetGPUDescriptorHandleForHeapStart());
    cbvHandle.Offset(cbvIndex, mCbvSrvUavDescriptorSize);

    cmdList->SetGraphicsRootDescriptorTable(0, cbvHandle);
    cmdList->DrawIndexedInstanced(ri->IndexCount, 1,
      ri->StartIndexLocation, ri->BaseVertexLocation, 0);
  }
}
```

在执行主要绘制函数Draw的过程中调用DrawRenderItems方法：

```
void ShapesApp::Draw(const GameTimer& gt)
{
  auto cmdListAlloc = mCurrFrameResource->CmdListAlloc;

  // 复用与记录命令有关的内存
  // 只有在GPU 执行完与该内存相关联的命令列表时，才能对此命令列表分配器进行重置
  ThrowIfFailed(cmdListAlloc->Reset());

  // 在通过ExecuteCommandList方法将命令列表添加到命令队列中之后，我们就可以对它
进行重置
  // 复用命令列表即复用与之相关的内存
  if(mIsWireframe)
  {
    ThrowIfFailed(mCommandList->Reset(
      cmdListAlloc.Get(), mPSOs["opaque_wireframe"].Get()));
  }
  else
  {
    ThrowIfFailed(mCommandList->Reset(cmdListAlloc.Get(),
    mPSOs["opaque"].Get()));
  }
```

```
  mCommandList->RSSetViewports(1, &mScreenViewport);
  mCommandList->RSSetScissorRects(1, &mScissorRect);

  // 根据资源的用途指示资源状态的转换
  mCommandList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER::Transition(CurrentBackBuffer(),
    D3D12_RESOURCE_STATE_PRESENT,
    D3D12_RESOURCE_STATE_RENDER_TARGET));

  // 清除后台缓冲区和深度缓冲区
  mCommandList->ClearRenderTargetView(CurrentBackBufferView(),
    Colors::LightSteelBlue, 0, nullptr);
  mCommandList->ClearDepthStencilView(DepthStencilView(),
    D3D12_CLEAR_FLAG_DEPTH | D3D12_CLEAR_FLAG_STENCIL,
    1.0f, 0, 0, nullptr);

  // 指定要渲染的目标缓冲区
  mCommandList->OMSetRenderTargets(1, &CurrentBackBufferView(),
    true, &DepthStencilView());

  ID3D12DescriptorHeap* descriptorHeaps[] = { mCbvHeap.Get() };
  mCommandList->SetDescriptorHeaps(_countof(descriptorHeaps), descriptorHe
aps);

  mCommandList->SetGraphicsRootSignature(mRootSignature.Get());

  int passCbvIndex = mPassCbvOffset + mCurrFrameResourceIndex;
  auto passCbvHandle = CD3DX12_GPU_DESCRIPTOR_HANDLE(
    mCbvHeap->GetGPUDescriptorHandleForHeapStart());
  passCbvHandle.Offset(passCbvIndex, mCbvSrvUavDescriptorSize);
  mCommandList->SetGraphicsRootDescriptorTable(1, passCbvHandle);

  DrawRenderItems(mCommandList.Get(), mOpaqueRitems);

  // 按照资源的用途指示资源状态的转换
  mCommandList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER::Transition(CurrentBackBuffer(),
    D3D12_RESOURCE_STATE_RENDER_TARGET,
    D3D12_RESOURCE_STATE_PRESENT));

  // 完成命令的记录
  ThrowIfFailed(mCommandList->Close());

  // 将命令列表加入到命令队列中用于执行
  ID3D12CommandList* cmdsLists[] = { mCommandList.Get() };
  mCommandQueue->ExecuteCommandLists(_countof(cmdsLists), cmdsLists);
```

```
    // 交换前后台缓冲区
    ThrowIfFailed(mSwapChain->Present(0, 0));
    mCurrBackBuffer = (mCurrBackBuffer + 1) % SwapChainBufferCount;

    // 增加围栏值，将之前的命令标记到此围栏点上
    mCurrFrameResource->Fence = ++mCurrentFence;

    // 向命令队列添加一条指令，以设置新的围栏点
    // GPU还在执行我们此前向命令队列中传入的命令，所以，GPU不会立即设置新
    // 的围栏点，这要等到它处理完Signal()函数之前的所有命令
    mCommandQueue->Signal(mFence.Get(), mCurrentFence);
}
```

7.6　细探根签名

我们在6.6.5节中曾介绍过根签名。根签名定义了：在绘制调用之前，需要绑定到渲染流水线上的资源，以及这些资源应如何映射到着色器的输入寄存器中。选择绑定到流水线上的资源要根据着色器程序的具体需求。从PSO被创建之时起，根签名和着色器程序的组合就开始生效。

7.6.1　根参数

回顾根签名的相关知识可知，根签名是由一系列根参数定义而成。到目前为止，我们只创建过存有一个描述符表的根参数。然而，根参数其实有3个类型可选。

1．描述符表（descriptor table）：描述符表引用的是描述符堆中的一块连续范围，用于确定要绑定的资源。

2．根描述符（root descriptor，又称为内联描述符，inline descriptor）：通过直接设置根描述符即可指示要绑定的资源，而且无需将它存于描述符堆中。但是，只有常量缓冲区的CBV，以及缓冲区的SRV/UAV（着色器资源视图/无序访问视图）才可以根描述符的身份进行绑定。这也就意味着纹理的SRV并不能作为根描述符来实现资源绑定。

3．根常量（root constant）：借助根常量可直接绑定一系列32位的常量值。

考虑到性能因素，可放入一个根签名的数据以64 DWORD为限。3种根参数类型占用空间的情况如下。

1．描述符表：每个描述符表占用1 DWORD。

2．根描述符：每个根描述符（64位的GPU虚拟地址）占用2 DWORD。

3．根常量：每个常量32位，占用1 DWORD。

我们可以创建出任意组合的根签名，只要它不超过64 DWORD的上限即可。根常量虽然用起来方便，但是它的空间消耗增加迅速。例如，倘若所用的“世界—视图—投影”矩阵只有常量数据，那么，我们就能用16个根常量来存储它，还无需再创建相应的常量缓冲区以及CBV堆。然而，这些根常量却“吃掉”了我们根签名预算的1/4。使用1个根描述符只占2 DWORD，而1个描述符表仅用1 DWORD。由于我们的应用程序会变得愈加复杂，其常量缓冲区中的数据也将越来越庞大，因而也不太可能仅使用根常量。所以，在现实的应用程序中，我们很可能会经常混用这3种根参数。

在代码中要通过填写CD3DX12_ROOT_PARAMETER结构体来描述根参数。就像我们之前曾见到过的以CD3DX作为前缀的其他辅助结构体一样，CD3DX12_ROOT_PARAMETER是对结构体D3D12_ROOT_PARAMETER进

行扩展，并增加一些辅助初始化函数而得来的[2]。

```
typedef struct D3D12_ROOT_PARAMETER
{
  D3D12_ROOT_PARAMETER_TYPE ParameterType;
  union
  {
    D3D12_ROOT_DESCRIPTOR_TABLE DescriptorTable;
    D3D12_ROOT_CONSTANTS Constants;
    D3D12_ROOT_DESCRIPTOR Descriptor;
  };
  D3D12_SHADER_VISIBILITY ShaderVisibility;
}D3D12_ROOT_PARAMETER;
```

1. `ParameterType`：下列枚举类型的成员之一，用于指示根参数的类型（描述符表、根常量、CBV根描述符、SRV根描述符、UAV根描述符）。

```
   enum D3D12_ROOT_PARAMETER_TYPE
   {
     D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE = 0,
     D3D12_ROOT_PARAMETER_TYPE_32BIT_CONSTANTS  = 1,
     D3D12_ROOT_PARAMETER_TYPE_CBV    = 2,
     D3D12_ROOT_PARAMETER_TYPE_SRV    = 3 ,
     D3D12_ROOT_PARAMETER_TYPE_UAV    = 4
   } D3D12_ROOT_PARAMETER_TYPE;
```

2. `DescriptorTable/Constants/Descriptor`：描述根参数的结构体。该联合体成员的填写依根签名的具体类型而定。我们会在7.6.2节、7.6.3节和7.6.4节中对此结构体展开讨论。

3. `ShaderVisibility`：下列枚举类型的成员之一，指定此根参数在着色器程序中的可见性。本书中一般会采用`D3D12_SHADER_VISIBILITY_ALL`枚举项。举个例子：如果知道某种

资源只会在像素着色器中使用，我们就可把此资源的这一项指定为D3D12_SHADER_VISIBILITY_PIXEL。限制根参数的可见性可能使程序的性能得到优化。

```
enum D3D12_SHADER_VISIBILITY
{
  D3D12_SHADER_VISIBILITY_ALL      = 0,
  D3D12_SHADER_VISIBILITY_VERTEX   = 1,
  D3D12_SHADER_VISIBILITY_HULL     = 2,
  D3D12_SHADER_VISIBILITY_DOMAIN   = 3,
  D3D12_SHADER_VISIBILITY_GEOMETRY = 4,
  D3D12_SHADER_VISIBILITY_PIXEL    = 5
}  D3D12_SHADER_VISIBILITY;
```

7.6.2　描述符表

通过填写D3D12_ROOT_PARAMETER结构体的成员DescriptorTable，即可进一步将根参数的类型定义为描述符表（descriptor table）。

```
typedef struct D3D12_ROOT_DESCRIPTOR_TABLE
{
  UINT NumDescriptorRanges;
  const D3D12_DESCRIPTOR_RANGE *pDescriptorRanges;
}    D3D12_ROOT_DESCRIPTOR_TABLE;
```

凭借上述结构体可以方便地指定一个D3D12_DESCRIPTOR_RANGE类型数组，以及该数组中元素的数量。

结构体D3D12_DESCRIPTOR_RANGE的定义如下：

```
typedef struct D3D12_DESCRIPTOR_RANGE
{
```

```
  D3D12_DESCRIPTOR_RANGE_TYPE RangeType;
  UINT NumDescriptors;
  UINT BaseShaderRegister;
  UINT RegisterSpace;
  UINT OffsetInDescriptorsFromTableStart;
}  D3D12_DESCRIPTOR_RANGE;
```

1. RangeType：下列枚举类型的成员之一，指示此范围（range）中的描述符类型：

```
   enum D3D12_DESCRIPTOR_RANGE_TYPE
   {
     D3D12_DESCRIPTOR_RANGE_TYPE_SRV   = 0,
     D3D12_DESCRIPTOR_RANGE_TYPE_UAV   = 1,
     D3D12_DESCRIPTOR_RANGE_TYPE_CBV   = 2 ,
     D3D12_DESCRIPTOR_RANGE_TYPE_SAMPLER = 3
   }   D3D12_DESCRIPTOR_RANGE_TYPE;
```

注意 Note

采样器描述符（sampler descriptor）将在与纹理贴图有关的章节里进行讨论。

---

2. NumDescriptors：范围内描述符的数量。

3. BaseShaderRegister：此描述符范围将要绑定到的基准着色器寄存器（base shader register）。比方说，如果您将NumDescriptors设为3，把BaseShaderRegister置为1，又令描述符范围的类型为常量缓冲区CBV，那么，这些资源将按以下方式与HLSL寄存器相绑定。

```
cbuffer cbA : register(b1) {...};
cbuffer cbB : register(b2) {...};
cbuffer cbC : register(b3) {...};
```

4. RegisterSpace：此属性将使您能够在不同的寄存器空间中指定着色器寄存器。例如，下列代码中的两种资源看起来似乎重复使用了寄存器槽t0，但实际上却并非如此，因为它们各自存在于不同的空间之中：

```
Texture2D gDiffuseMap : register(t0, space0);
Texture2D gNormalMap  : register(t0, space1);
```

如果在着色器中没有显式地指定空间寄存器，那么它将自动默认为space0。一般情况下我们通常使用space0，但是对于资源数组来说，使用多重寄存器空间会更加方便，尤其是在数组大小未知的情况下，更是如此。

5. OffsetInDescriptorsFromTableStart：此描述符范围（range of descriptor）距离描述符表起始地址的偏移量。详见以下示例。

由于我们可能将各种类型的描述符混合放置在一个描述符表中，所以会把寄存器槽参数初始化为：一个存有一系列D3D12_DESCRIPTOR_RANGE实例的描述符表。假设我们以“2个CBV，3个SRV和1个UAV”这3个描述符范围的顺序指定了由6个描述符构成的表，则此表的定义为：

```
// 用2个CBV，3个SRV和1个UAV来创建一个描述符表
CD3DX12_DESCRIPTOR_RANGE descRange[3];
```

```
descRange[0].Init(
  D3D12_DESCRIPTOR_RANGE_TYPE_CBV, // 描述符的类型
  2, // 描述符的个数
  0, // 此根参数将要绑定到的基准着色器寄存器
  0, // 寄存器空间
  0);// 到此描述表起始地址的偏移量
descRange[1].Init(
  D3D12_DESCRIPTOR_RANGE_TYPE_SRV, // 描述符的类型
  3, // 描述符的个数
  0, // 此根参数将要绑定到的基准着色器寄存器
  0, // 寄存器空间
  2);// 到此描述表起始地址的偏移量
descRange[2].Init(
  D3D12_DESCRIPTOR_RANGE_TYPE_UAV, // 描述符的类型
  1, // 描述符的个数
  0, // 此根参数将要绑定到的基准着色器寄存器
  0, // 寄存器空间
  5);// 到此描述符表起始地址的偏移量
slotRootParameter[0].InitAsDescriptorTable(
  3, descRange, D3D12_SHADER_VISIBILITY_ALL);
```

像之前所见的其他辅助结构体一样，CD3DX12_DESCRIPTOR_RANGE继承自结构体D3D12_DESCRIPTOR_RANGE，我们将使用其中的下列初始函数：

```
void CD3DX12_DESCRIPTOR_RANGE::Init(
    D3D12_DESCRIPTOR_RANGE_TYPE rangeType,
    UINT numDescriptors,
    UINT baseShaderRegister,
    UINT registerSpace = 0,
    UINT offsetInDescriptorsFromTableStart =
    D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND);
```

上面配置的描述符表共存有6个描述符，而应用程序期望绑定描述符堆中一块连续的描述符范围，其中依次包含2个CBV、3个SRV和1个UAV。我们可以看到所有类型的描述符表都是以寄存器0作为基准寄存器（baseShaderRegister），但是却并没有发生寄存器“重叠（overlap）”冲突，这是因为CBV、SRV和UAV都分别被绑定在不同类

型的寄存器上，并始于各自的寄存器0。

我们能通过指定D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND，令Direct3D来计算OffsetInDescriptorsFromTableStart的值。这将使Direct3D根据表中前一个描述符范围中描述符的数量来计算偏移量。可以看出，CD3DX12_DESCRIPTOR_RANGE::Init方法默认的寄存器空间为0，且参数OffsetInDescriptorsFromTableStart的默认值为D3D12_DESCRIPTOR_RANGE_ OFFSET_APPEND。

7.6.3　根描述符

通过填写结构体D3D12_ROOT_PARAMETER中的成员Descriptor，即可将根参数的类型进一步定义为根描述符（root descriptor）。

```
typedef struct D3D12_ROOT_DESCRIPTOR
{
  UINT ShaderRegister;
  UINT RegisterSpace;
}D3D12_ROOT_DESCRIPTOR;
```

1. ShaderRegister：此描述符将要绑定的着色器寄存器。例如，如果将它指定为2，而且此根参数是一个CBV，则此根参数将被映射到register(b2)中的常量缓冲区：

```
   cbuffer cbPass : register(b2) {...};
```

2. RegisterSpace：参见D3D12_DESCRIPTOR_RANGE::RegisterSpace。

与描述符表需要在描述符堆中设置对应的描述符句柄不同，要配置根描述符，我们只需简单而又直接地绑定资源的虚拟地址即可。

```
UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof
  (ObjectConstants));

D3D12_GPU_VIRTUAL_ADDRESS objCBAddress = objectCB->GetGPUVirtualAddress();

// 偏移到缓冲区中此物体常量的地址
objCBAddress += ri->ObjCBIndex*objCBByteSize;

cmdList->SetGraphicsRootConstantBufferView(
   0, // 根参数索引，即将当前根描述符绑定到此编号的寄存器槽位
   objCBAddress);[3]
```

7.6.4　根常量

通过填写结构体D3D12_ROOT_PARAMETER的成员Constants，即可进一步将根参数的类型定义为根常量（root constant）。

```
  typedef struct D3D12_ROOT_CONSTANS
  {
    UINT ShaderRegister;
    UINT RegisterSpace;
    UINT Num32BitValues;
  }   D3D12_ROOT_CONSTANTS;
```

1. ShaderRegister：参见D3D12_ROOT_DESCRIPTOR::ShaderRegister。

2. RegisterSpace：参见D3D12_DESCRIPTOR_RANGE::RegisterSpace。

3. Num32BitValues：此值为根参数所需的32位常量的个数。

设置根常量仍要将数据映射到着色器视角中的常量缓冲区内，下面是此过程的详细示例：

```
// 应用程序代码部分：根签名的定义
CD3DX12_ROOT_PARAMETER slotRootParameter[1];
slotRootParameter[0].InitAsConstants(12, 0);

// 根签名即是一系列根参数
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(1, slotRootParameter,
  0, nullptr,
  D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

// 应用程序代码部分：将根常量设置到寄存器b0
auto weights = CalcGaussWeights(2.5f);
int blurRadius = (int)weights.size() / 2;

cmdList->SetGraphicsRoot32BitConstants(0, 1, &blurRadius, 0);
cmdList->SetGraphicsRoot32BitConstants(0, (UINT)weights.size(),
    weights.data(), 1);

// HLSL代码部分
cbuffer cbSettings : register(b0)
{
    // 我们无法获取常量缓冲区中映射有根常量数据的数组元素，所以只得将每个元素分别单独列出

    int gBlurRadius;

    // 最多支持11种模糊权值（blur weight）
    float w0;
    float w1;
    float w2;
    float w3;
    float w4;
    float w5;
    float w6;
    float w7;
    float w8;
    float w9;
    float w10;
};
```

ID3D12GraphicsCommandList::SetGraphicsRoot32BitConstants方法的原型如下：

```
void ID3D12GraphicsCommandList::SetGraphicsRoot32BitConstants(
  UINT RootParameterIndex,
  UINT Num32BitValuesToSet,
  const void *pSrcData,
  UINT DestOffsetIn32BitValues);
```

1. RootParameterIndex：我们所设置的根参数的索引，即将根常量绑定到此槽号的寄存器。

2. Num32BitValuesToSet：本次设置的32位常量数据的个数。

3. pSrcData：指向将要设置的32位常量数据数组的指针。

4. DestOffsetIn32BitValues：本次设置的第一个常量数据在常量缓冲区中的偏移量（用一个32位数表示）。

如同根描述符一样，设置根常量时无需涉及描述符堆。

7.6.5　更复杂的根签名示例

思考一下着色器需要下列资源的情景：

```
Texture2D gDiffuseMap : register(t0);

cbuffer cbPerObject : register(b0)
{
  float4x4 gWorld;
  float4x4 gTexTransform;
};
```

```
cbuffer cbPass : register(b1)
{
  float4x4 gView;
  float4x4 gInvView;
  float4x4 gProj;
  float4x4 gInvProj;
  float4x4 gViewProj;
  float4x4 gInvViewProj;
  float3 gEyePosW;
  float cbPerObjectPad1;
  float2 gRenderTargetSize;
  float2 gInvRenderTargetSize;
  float gNearZ;
  float gFarZ;
  float gTotalTime;
  float gDeltaTime;
  float4 gAmbientLight;
  Light gLights[MaxLights];
};

cbuffer cbMaterial : register(b2)
{
  float4  gDiffuseAlbedo;
  float3  gFresnelR0;
  float  gRoughness;
  float4x4 gMatTransform;
};
```

此着色器对应的根签名描述如下：

```
CD3DX12_DESCRIPTOR_RANGE texTable;
  texTable.Init(
  D3D12_DESCRIPTOR_RANGE_TYPE_SRV,
  1,  // 根签名的数量
  0); // 寄存器t0

  // 根参数可以是描述符表，根描述符或根常量
  CD3DX12_ROOT_PARAMETER slotRootParameter[4];

  // 性能小提示:按变更频率高->低的顺序进行排列
  slotRootParameter[0].InitAsDescriptorTable(1,
    &texTable, D3D12_SHADER_VISIBILITY_PIXEL);
  slotRootParameter[1].InitAsConstantBufferView(0); // 寄存器b0
  slotRootParameter[2].InitAsConstantBufferView(1); // 寄存器b1
  slotRootParameter[3].InitAsConstantBufferView(2); // 寄存器b2
```

```
// 根签名就是一系列根参数
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(4, slotRootParameter,
  0, nullptr,
D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
```

7.6.6 根参数的版本控制

根实参（root argument，其实有些文献也将此翻译为“根参数”，为示区别在此译为“根实参”）即我们向根参数传递的实际数值。考虑下列代码，其中，我们在每次绘制调用之间都修改了根实参（此例中只有描述符表）：

```
for(size_t i = 0; i < mRitems.size(); ++i)
{
  const auto& ri = mRitems[i];

  ...

  // 偏移到此帧中渲染项的CBV
  int cbvOffset = mCurrFrameResourceIndex*(int)mRitems.size();
  cbvOffset += ri.CbIndex;
  cbvHandle.Offset(cbvOffset, mCbvSrvDescriptorSize);

  // 指定此次绘制调用所需的描述符
  cmdList->SetGraphicsRootDescriptorTable(0, cbvHandle);

  cmdList->DrawIndexedInstanced(
    ri.IndexCount, 1,
    ri.StartIndexLocation,
    ri.BaseVertexLocation, 0);
}
```

在每次执行绘制调用时，将使用针对当前绘制调用所设置的根实参状态。这个工作得以实现全然是因为：硬件会自动为每次绘制调用保存根实参当时状态的快照（snapshot）。换句话说，系统会为每次绘制调

用而自动对根参数进行版本控制。

值得注意的是，根签名可以为着色器提供比实际所用更多的字段。例如，如果根签名在根参数2中指定了一个根CBV，但是着色器却根本不使用该常量缓冲区，然而，只要根签名为着色器传递了所有必备的数据，则这个设置组合就是合法的。

考虑到性能因素，我们应当使根签名尽可能小。其中一个原因就是在每次绘制调用时，根实参都会自动控制版本。而根签名越大，则根实参的快照也就越大。另外，SDK文档建议：根签名中的根参数应当按照变更频率，由高至低排列。Direct3D 12文档也建议我们尽可能避免频繁切换根签名。因此，一个好的办法就是令您创建的多个PSO共享同一个根签名。特别是，有时我们可以得益于多个着色器程序采用一个“超级”根签名的模式，哪怕并不是所有的着色器都能充分利用该根签名中定义的全部参数。但从另一方面来讲，我们也要考虑此“超级”根签名为了实际需要所必须达到的规模。如果它过于庞大，则其无需切换根签名所带来的优势可能就会因此而被抵消。

7.7　陆地与波浪演示程序

在本小节中，我们将展示如何构建图7.7所示的“Land and Waves”（陆地与波浪）演示程序。此示例构建了一个三角形栅格（grid），并通过将其中的顶点偏移到不同的高度来创建地形（terrain）。另外，还要使用另一个三角形栅格来表现水，动态的改变其顶点高度来创建波浪。此例程还将针对不同的常量缓冲区而切换所用的根描述符，这使我们能够摆脱设置琐琐的CBV描述符堆。

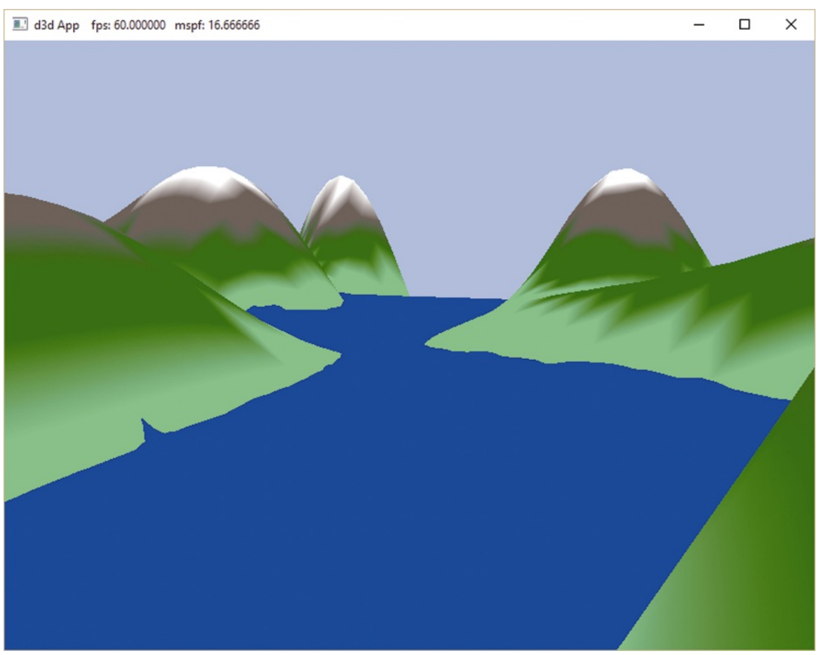


图7.7　“Land and Waves”演示程序的效果。由于我们还未运用光照技术，所以很难看出水面的涟漪起伏。此时，按下“1”键以线框模式来观察场景不失为一种观看水面波动的好方法

实值函数$y=f(x,y)$的图像是一个曲面。我们可以通过在xz平面内构造一个栅格来近似地表示这个曲面，其中的每个四边形都是由两个三角形所构成的，接下来再利用此函数计算出每个栅格点处的高度即可，如

图7.8所示。

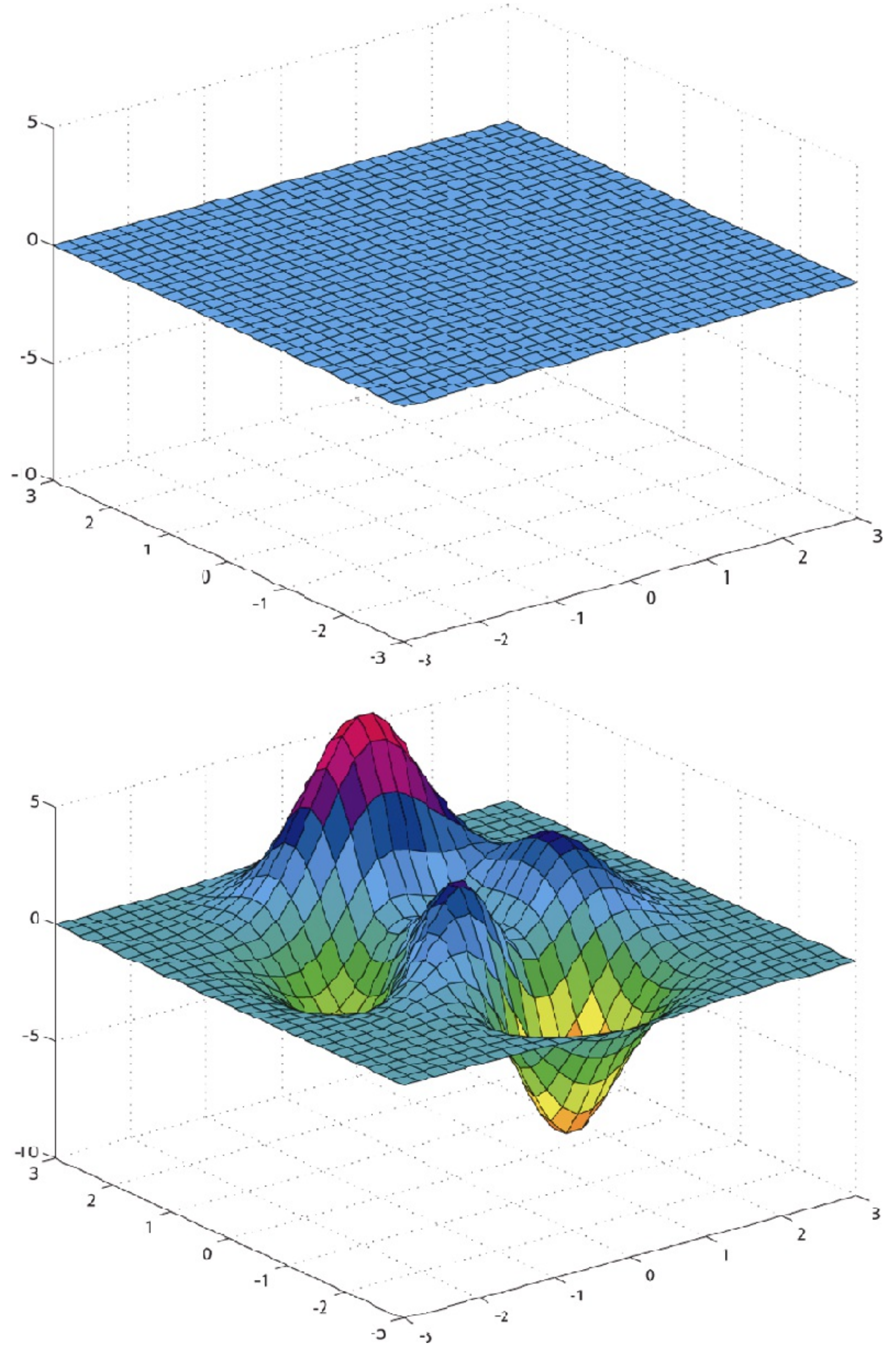


图7.8　（上图）首先在xz平面内“铺设”一层栅格。（下图）运用函数$f(x, z)$为每个栅格点获取其对应的y坐标。再借助$(x, f(x, z), z)$诸点即可绘制出曲面的图像

7.7.1 生成栅格顶点

由以上分析可知，我们主要的任务就是来构建xz平面内的栅格。$m \times n$个顶点所构成的栅格具有$(m-1)\times(n-1)$个四边形（或称单元格），如图7.9所示。每个单元格由两个三角形组成，即有共$2\cdot(m-1)\times(n-1)$个三角形。如果栅格的宽为w且深度为d，则单元格在x轴与z轴方向上的间距分别为$dx = w/(n-1)$与$dz = d/(m-1)$。为了生成栅格顶点，我们从左上角开始，逐行渐进地计算顶点坐标。在xz平面中，第i行、第j列栅格顶点的坐标可以表示为：

$$v_{ij} = [-0.5w + j\cdot dx, \quad 0.0, \quad 0.5d - i\cdot dz]$$

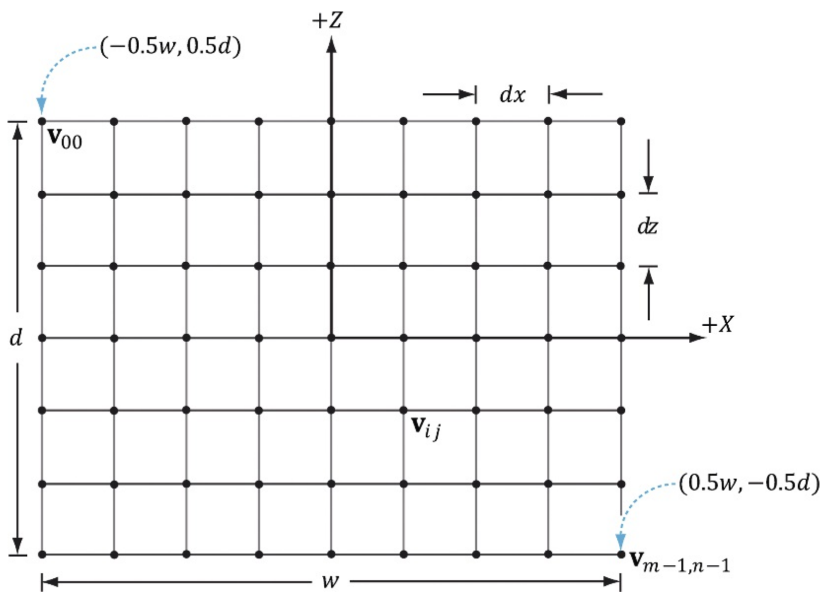


图7.9　栅格的结构

下列代码用于生成栅格顶点：

```
GeometryGenerator::MeshData
GeometryGenerator::CreateGrid(float width, float depth, uint32 m, uint32 n
)
```

```
{
  MeshData meshData;

  uint32 vertexCount = m*n;
  uint32 faceCount  = (m-1)*(n-1)*2;

  float halfWidth = 0.5f*width;
  float halfDepth = 0.5f*depth;

  float dx = width / (n-1);
  float dz = depth / (m-1);

  float du = 1.0f / (n-1);
  float dv = 1.0f / (m-1);

  meshData.Vertices.resize(vertexCount);
  for(uint32 i = 0; i < m; ++i)
  {
    float z = halfDepth - i*dz;
    for(uint32 j = 0; j < n; ++j)
    {
      float x = -halfWidth + j*dx;

      meshData.Vertices[i*n+j].Position = XMFLOAT3(x, 0.0f, z);
      meshData.Vertices[i*n+j].Normal  = XMFLOAT3(0.0f, 1.0f, 0.0f);
      meshData.Vertices[i*n+j].TangentU = XMFLOAT3(1.0f, 0.0f, 0.0f);

      // 在栅格上拉伸纹理。
      meshData.Vertices[i*n+j].TexC.x = j*du;
      meshData.Vertices[i*n+j].TexC.y = i*dv;
    }
  }
```

7.7.2　生成栅格索引

完成顶点的计算之后，我们需要通过指定索引来定义栅格三角形。为此，我们再次从左上角出发，逐行遍历每个四边形，并计算索引以定义构成每个四边形的两个三角形。参考图7.10可以发现，对于一个规模为$m \times n$顶点的栅格来说，四边形中两个三角形的线性数组索引的计算

方法如下：

$$\Delta ABC = (i\cdot n+j, \quad i\cdot n+j+1, \quad (i+1)\cdot n+j)$$
$$\Delta CBD = ((i+1)\cdot n+j, \quad i\cdot n+j+1, \quad (i+1)\cdot n+j+1)$$

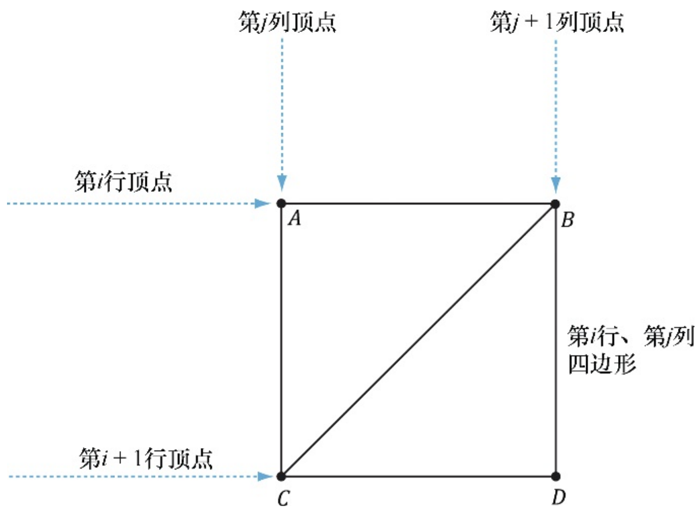


图7.10　第i行、第j列四边形的顶点索引

相应的代码为：

```
meshData.Indices32.resize(faceCount*3); // 每个三角形面有3个索引

// 遍历每个四边形并计算索引
uint32 k = 0;
for(uint32 i = 0; i < m-1; ++i)
{
  for(uint32 j = 0; j < n-1; ++j)
  {
    meshData.Indices32[k]   = i*n+j;
    meshData.Indices32[k+1] = i*n+j+1;
    meshData.Indices32[k+2] = (i+1)*n+j;

    meshData.Indices32[k+3] = (i+1)*n+j;
    meshData.Indices32[k+4] = i*n+j+1;
    meshData.Indices32[k+5] = (i+1)*n+j+1;

    k += 6; // 下一个四边形
  }
}
```

```
  return meshData;
}
```

7.7.3　应用计算高度的函数

待栅格创建完成后，我们就能从MeshData栅格中获取所需的顶点元素，根据顶点的高度（y坐标）将平坦的栅格变为用于表现山峰起伏的曲面，并为它生成对应的颜色。

```
// 请不要与GeometryGenerator::Vertex结构体相混淆
struct Vertex
{
  XMFLOAT3 Pos;
  XMFLOAT4 Color;
};
void LandAndWavesApp::BuildLandGeometry()
{
  GeometryGenerator geoGen;
  GeometryGenerator::MeshData grid = geoGen.CreateGrid(160.0f, 160.0f, 50,
 50);

  //
  // 获取我们所需要的顶点元素，并利用高度函数计算每个顶点的高度值
  // 另外，顶点的颜色要基于它们的高度而定
  // 所以，图像中才会有看起来如沙质的沙滩、山腰处的植被以及山峰处的积雪
  //

  std::vector<Vertex> vertices(grid.Vertices.size());
  for(size_t i = 0; i < grid.Vertices.size(); ++i)
  {
    auto& p = grid.Vertices[i].Position;
    vertices[i].Pos = p;
    vertices[i].Pos.y = GetHillsHeight(p.x, p.z);

    // 基于顶点高度为它上色
    if(vertices[i].Pos.y < -10.0f)
    {
      // 沙滩的颜色
      vertices[i].Color = XMFLOAT4(1.0f, 0.96f, 0.62f, 1.0f);
    }
    else if(vertices[i].Pos.y < 5.0f)
```

```
    {
      // 浅黄绿色
      vertices[i].Color = XMFLOAT4(0.48f, 0.77f, 0.46f, 1.0f);
    }
    else if(vertices[i].Pos.y < 12.0f)
    {
      // 深黄绿色
      vertices[i].Color = XMFLOAT4(0.1f, 0.48f, 0.19f, 1.0f);
    }
    else if(vertices[i].Pos.y < 20.0f)
    {
      // 深棕色
      vertices[i].Color = XMFLOAT4(0.45f, 0.39f, 0.34f, 1.0f);
    }
    else
    {
      // 白雪皑皑
      vertices[i].Color = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);
    }
  }

  const UINT vbByteSize = (UINT)vertices.size() * sizeof(Vertex);

  std::vector<std::uint16_t> indices = grid.GetIndices16();
  const UINT ibByteSize = (UINT)indices.size() * sizeof(std::uint16_t);

  auto geo = std::make_unique<MeshGeometry>();
  geo->Name = "landGeo";

  ThrowIfFailed(D3DCreateBlob(vbByteSize, &geo->VertexBufferCPU));
  CopyMemory(geo->VertexBufferCPU->GetBufferPointer(), vertices.data(),
    vbByteSize);

  ThrowIfFailed(D3DCreateBlob(ibByteSize, &geo->IndexBufferCPU));
  CopyMemory(geo->IndexBufferCPU->GetBufferPointer(), indices.data(),
    ibByteSize);

  geo->VertexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
    mCommandList.Get(), vertices.data(), vbByteSize, geo->VertexBufferUplo
ader);

  geo->IndexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
    mCommandList.Get(), indices.data(), ibByteSize,
    geo->IndexBufferUploader);

  geo->VertexByteStride = sizeof(Vertex);
  geo->VertexBufferByteSize = vbByteSize;
```

```
  geo->IndexFormat = DXGI_FORMAT_R16_UINT;
  geo->IndexBufferByteSize = ibByteSize;

  SubmeshGeometry submesh;
  submesh.IndexCount = (UINT)indices.size();
  submesh.StartIndexLocation = 0;
  submesh.BaseVertexLocation = 0;

  geo->DrawArgs["grid"] = submesh;

  mGeometries["landGeo"] = std::move(geo);
}
```

在此演示程序中使用的函数$f(x,z)$为：

```
float LandAndWavesApp::GetHillsHeight(float x, float z)const
{
  return 0.3f*(z*sinf(0.1f*x) + x*cosf(0.1f*z));
}
```

最后得到的图像看起来就似山川地形（参见图7.7）。

7.7.4　根常量缓冲区视图

此“Land and Waves”演示程序较之前“Shape”例程的另一个区别是：我们在前者中使用了根描述符，因此就可以摆脱描述符堆而直接绑定CBV了。为此，程序还要做如下改动：

1．根签名需要变为取两个根CBV，而不再是两个描述符表。

2．不采用CBV堆，更无需向其填充描述符。

3．涉及一种用于绑定根描述符的新语法。

新的根签名定义如下：

```
// 根参数可以是描述符表，根描述符或根常量
CD3DX12_ROOT_PARAMETER slotRootParameter[2];

// 创建根CBV。
slotRootParameter[0].InitAsConstantBufferView(0); // 物体的CBV
slotRootParameter[1].InitAsConstantBufferView(1); // 渲染过程CBV

// 根签名即是一系列根参数的组合
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(2, slotRootParameter, 0,
nullptr, D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
```

由此可以看出，我们使用辅助方法InitAsConstantBufferView来创建根CBV，并通过其参数来指定此根参数将要绑定的着色器寄存器（在上面的示例中，指定的着色器常量缓冲区寄存器分别为“b0”和“b1”）。

现在，让我们利用下列方法，以传递参数的方式将CBV与某个根描述符相绑定：

```
void
ID3D12GraphicsCommandList::SetGraphicsRootConstantBufferView(
  UINT RootParameterIndex,
  D3D12_GPU_VIRTUAL_ADDRESS BufferLocation);
```

1. RootParameterIndex：CBV将要绑定到的根参数索引，即寄存器的槽位号。

2. BufferLocation：含有常量缓冲区数据资源的虚拟地址。

经过此次变化，我们的绘制代码现在看起来是这样的：

```
void LandAndWavesApp::Draw(const GameTimer& gt)
```

```
{
  [...]

  // 绑定渲染过程中所用的常量缓冲区。在每个渲染过程中，这段代码只需执行一次
  auto passCB = mCurrFrameResource->PassCB->Resource();
  mCommandList->SetGraphicsRootConstantBufferView(1, passCB->
    GetGPUVirtualAddress());

  DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Opaque
]);

  [...]
}

void LandAndWavesApp::DrawRenderItems(
  ID3D12GraphicsCommandList* cmdList,
  const std::vector<RenderItem*>& ritems)
{
  UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof
    (ObjectConstants));

  auto objectCB = mCurrFrameResource->ObjectCB->Resource();

  // 对于每个渲染项来说……
  for(size_t i = 0; i < ritems.size(); ++i)
  {
    auto ri = ritems[i];

    cmdList->IASetVertexBuffers(0, 1, &ri->Geo->VertexBufferView());
    cmdList->IASetIndexBuffer(&ri->Geo->IndexBufferView());
    cmdList->IASetPrimitiveTopology(ri->PrimitiveType);

    D3D12_GPU_VIRTUAL_ADDRESS objCBAddress = objectCB->GetGPUVirtualAddres
s();
    objCBAddress += ri->ObjCBIndex*objCBByteSize;

    cmdList->SetGraphicsRootConstantBufferView(0, objCBAddress);

    cmdList->DrawIndexedInstanced(ri->IndexCount, 1,
      ri->StartIndexLocation, ri->BaseVertexLocation, 0);
  }
}
```

7.7.5 动态顶点缓冲区

到目前为止，我们始终都将顶点数据存于默认的缓冲区资源当中，可借此存储静态几何体。这就是说，我们不能动态地改变此资源中所存的几何体——只能一次性设置好数据，再以GPU读取其中的数据并进行绘制。此时，一种名为动态顶点缓冲区（dynamic vertex buffer）的资源应运而生，它允许用户频繁地更改其中的顶点数据。比如说，我们可在每一帧都修改其中的顶点数据。现假设我们正在模拟波浪的运动，第一步就是要求出函数$f(x,z,t)$此波浪方程的解。该函数表示的是：在t时刻，位于xz平面内每个点处的波浪高度。如果采用这个函数来绘制波浪，我们应使用之前绘制山峰和谷地时所用的三角形栅格，利用此函数$f(x,z,t)$针对每个栅格点进行计算，以此求出位于每个栅格点处的波浪高度。由于该函数依赖于时间t（即波面随时间发生变化），我们就需要在短时间内（比如说每1/30秒）多次调用此函数，从而获得平滑的动画效果。这样一来，我们便需要创建一个动态顶点缓冲区，随着时间的流走而更新三角形栅格顶点的高度。另一种要用到动态顶点缓冲区的情况是：需要执行复杂的物理模拟计算和碰撞检测（collision detection）的粒子系统（particle system）。在该系统中，为了找寻每个粒子的新位置，我们在每一帧都要用CPU进行物理模拟计算以及碰撞检测。由于粒子的位置在每一帧都会有所变化，因此，我们就需要在绘制每一帧时借助动态顶点缓冲区来更新粒子的位置。

在通过上传缓冲区来更新常量缓冲区中的数据时，我们已经接触过由CPU在每一帧向GPU上传数据的具体流程。使用之前编写的`UploadBuffer`类即可重施故技，但是这次存储的资源是顶点数组，而非常量缓冲区数组：

```
  std::unique_ptr<UploadBuffer<Vertex>> WavesVB = nullptr;

  WavesVB = std::make_unique<UploadBuffer<Vertex>>(
    device, waveVertCount, false);
```

由于我们在每一帧都要从CPU向波浪动态顶点缓冲区上传新的数据内容，所以需要将动态顶点缓冲区存为一种帧资源。若非如此，我们就有可能在GPU未完成最近一帧的处理之前就覆写了相关内存中的数据。

在每一帧中，我们都以下列方式来模拟波浪并更新顶点缓冲区：

```
void LandAndWavesApp::UpdateWaves(const GameTimer& gt)
{
  // 每隔1/4秒就要生成一个随机波浪
  static float t_base = 0.0f;
  if((mTimer.TotalTime() - t_base) >= 0.25f)
  {
    t_base += 0.25f;

    int i = MathHelper::Rand(4, mWaves->RowCount() - 5);
    int j = MathHelper::Rand(4, mWaves->ColumnCount() - 5);

    float r = MathHelper::RandF(0.2f, 0.5f);

    mWaves->Disturb(i, j, r);
  }

  // 更新模拟的波浪
  mWaves->Update(gt.DeltaTime());

  // 用波浪方程求出的新数据来更新波浪顶点缓冲区
  auto currWavesVB = mCurrFrameResource->WavesVB.get();
  for(int i = 0; i < mWaves->VertexCount(); ++i)
  {
    Vertex v;

    v.Pos = mWaves->Position(i);
    v.Color = XMFLOAT4(DirectX::Colors::Blue);

    currWavesVB->CopyData(i, v);
  }
```

```
  // 将波浪渲染项的动态顶点缓冲区设置到当前帧的顶点缓冲区
  mWavesRitem->Geo->VertexBufferGPU = currWavesVB->Resource();
}
```

注意 Note

我们保存了一份波浪渲染项的引用（mWavesRitem），从而可以动态地调整其顶点缓冲区。由于渲染项的顶点缓冲区是个动态的缓冲区，并且每一帧都在发生改变，因此这样做很有必要。

在使用动态缓冲区时会不可避免的产生一些开销，因为必须将新数据从CPU端内存回传至GPU端显存。这样说来，如果静态缓冲区能实现相同的工作，那么应当比动态缓冲区更受青睐。Direct3D在最近的版本中已经引入新的特性，以减少对动态缓冲区的需求。例如：

1．简单的动画可以在顶点着色器中实现。

2．可以使用渲染到纹理（render to texture），或者计算着色器（compute shader）与顶点纹理拾取（vertex texture fetch）等技术来实现上述波浪模拟，而且全程皆是在GPU中进行的。

3．几何着色器为GPU创建或销毁图元提供了支持，在几何着色器出现之前，一般用CPU来处理相关任务。

4．在曲面细分阶段中可以通过GPU来对几何体进行镶嵌化处理，

在硬件曲面细分出现之前，通常用CPU来处理相关任务。

我们也可以用动态缓冲区来创建索引缓冲区。然而，在“Land and Waves”演示程序中，由于三角形的拓扑结构保持不变，而仅修改了顶点的高度，因此只需将顶点缓冲区设置为动态缓冲区即可。

本章中的“Land and Waves”例程通过一个动态顶点缓冲区实现了本节开端所描述的简易波浪模拟。对于本书来说，我们并没有涉及波浪模拟的具体算法细节（对此请见[Lengyel02]），而是把重心更多地放在与动态缓冲区有关的处理流程之上：即用CPU来更新波浪的模拟数据，再通过上传缓冲区更新顶点数据。

注意 Note

再次重申：此演示程序也可以通过如渲染到纹理，或者计算着色器以及顶点纹理拾取等高级技术在GPU上加以实现。由于我们现在还未讲到这些主题，所以此波浪模拟程序依然要在CPU上运行，并借助动态顶点缓冲区来更新顶点数据。

7.8　小结

1．在每帧中等待GPU处理完队列中所有命令的做法效率极低，因为这种策略在某些时刻会导致CPU或GPU处于空闲状态。一种更有效的技巧是创建帧资源（frame resource）——一个由每帧都需CPU来修改的资源所构成的环形数组。这种方法令CPU无需等待GPU结束当前的任务，即可继续处理下一帧的相关工作；对此，CPU只需处理下一个可用的（即GPU没在使用中的）帧资源。如果CPU处理帧的速度总是快于GPU，则CPU必在某些时刻等待GPU追赶上来，但此情景又正是我们所期盼的：不仅GPU的处理能力将得到充分的发挥，同时，多出来的CPU资源又总是可被游戏的其他部分，如AI，物理模拟与游戏逻辑所利用。

2．我们可以用`ID3D12DescriptorHeap::GetCPUDescriptorHandleForHeapStar`方法来获取堆中第一个描述符的句柄，通过`ID3D12Device::GetDescriptorHandleIncrementSize`方法得到描述符的大小（依赖于硬件与描述符的类型）。一旦知道了描述符增量的大小，我们就能用两种`CD3DX12_CPU_DESCRIPTOR_HANDLE::Offset`方法之一偏移至第n个描述符的句柄处：

```
// 指定要偏移到的描述符的编号，再将它乘以描述符的增量大小
D3D12_CPU_DESCRIPTOR_HANDLE handle = mCbvHeap->
    GetCPUDescriptorHandleForHeapStart();
handle.Offset(n * mCbvSrvDescriptorSize);

// 或者用另一种等价实现，先指定要偏移到的描述符编号，再设置描述符的增量大小
D3D12_CPU_DESCRIPTOR_HANDLE handle = mCbvHeap->
```

```
    GetCPUDescriptorHandleForHeapStart();
handle.Offset(n, mCbvSrvDescriptorSize);
```

CD3DX12_GPU_DESCRIPTOR_HANDLE类型有着同样的偏移方法。

3．根签名定义了在绘制调用开始之前，需要与渲染流水线相绑定的资源，以及这些资源将被映射到的具体着色器输入寄存器。绑定到流水线的具体资源要根据着色器程序来确定。在创建PSO后，根签名与着色器程序的组合就开始生效了。根签名由一系列根参数所构成。根参数可以是描述符表、根描述符或根常量。描述符表在堆中指定了一块描述符的连续范围。根描述符用于直接绑定根签名中的描述符（此过程无需涉及描述符堆）。而根常量则用于直接绑定根签名中的常量数据。出于性能的原因，1个根签名中所能容纳的数据大小被限制为最多64 DWORD。每个描述符表占1 DWORD，每个根描述符用2 DWORD，而每个32位的根常量占用1 DWORD。硬件会为每次绘制调用而自动保存根实参的快照。这样一来，我们就能在每次绘制调用的过程中安全地修改根实参了。但是，我们也应当尽量缩小根签名的规模，以此降低内存间数据的复制量。

4．当顶点缓冲区的内容在运行时需要频繁更新（比如在每一帧，或每1/30秒就要更新一次），动态顶点缓冲区就派上了用场。我们可以使用UploadBuffer类来实现动态顶点缓冲区，但这次存储的是顶点数组，而非常量缓冲区数组。由于我们在每一帧都要从CPU向波浪动态顶点缓冲区上传新数据，所以需要将动态顶点缓冲区存为一种帧资源。在使用动态顶点缓冲区的过程中，难免会产生一些开销，这是因为新数据必将从CPU端内存回传至GPU端的显存。因此，在静态顶点缓冲区也可

以胜任相同工作的情况下，它会比动态顶点缓冲区更受青睐。对此，Direct3D的最新版本已经引进了一些新的特性，以减少动态缓冲区的使用。

7.9 练习

1．修改“Shapes”例程，以`GeometryGenerator::CreateGeosphere`方法替换程序中的`GeometryGenerator::CreateSphere`方法，并分别尝试使用0、1、2、3这4种细分等级。

2．修改“Shapes”演示程序，用16个根常量来取代描述符表，以此设置物体的世界矩阵。

3．在本书配套资源中，有一个名为Models/Skull.txt的文件。此文件含有渲染图7.11中骷髅头所需的顶点列表和索引列表。通过使用像记事本[4]这样的文本编辑器来查阅此文件，并修改“Shapes”演示程序来加载和渲染此骷髅头网格。

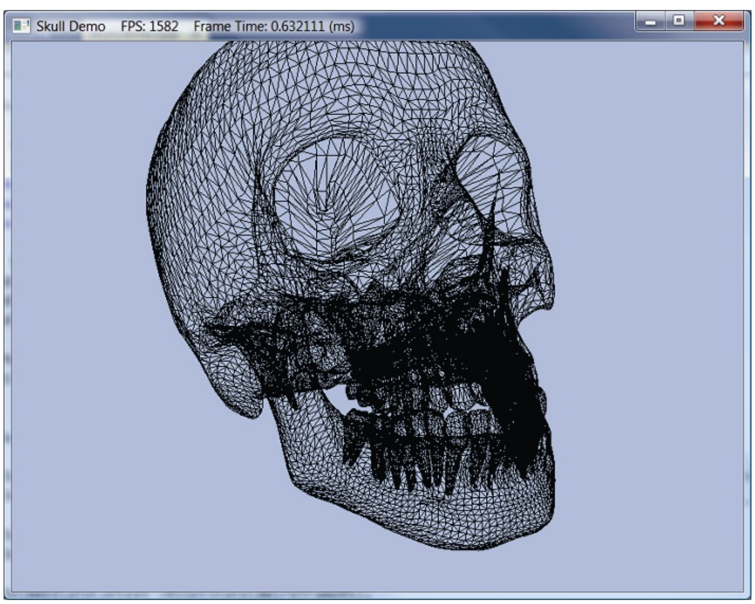


图7.11　练习4的渲染输出效果

[1] 预览版中ID3D12CommandQueue接口的GetLastCompletedFence()与SetEventOnFenceCompletion()方法，在正式版里已取消，并由ID3D12Fence接口的GetCompletedValue()与SetEventOnCompletion()方法替代。

[2] D3D12_ROOT_PARAMETER用于描述根签名1.0，而根签名1.1则用D3D12_ROOT_PARAMETER1来加以描述。

[3] 此函数的细节请参见7.7.4节。

[4] 对这种大数据文件感兴趣的话，最好不要用记事本，上兆的文本开起来要花若干秒。推荐尝试sublime，或直接在VS里开启。

第8章　光照

我们先来考虑图8.1，其中左侧是不受光照的球体，右侧是受到光照的球体。正如我们所见，左侧的球体看起来过于扁平——似乎完全不像是一个球体，而仅是一个2D圆片。而右侧的球体则看起来是颇具立体效果——光照（lighting）和阴影（shading）不仅使我们感受到目标物体的实体形状，还展现出了它的体积感。事实上，我们在视觉上对世界的感知依靠的是光照及其与材质（material）的交互。因此，在生成逼真场景的众多问题之中，首先要解决的就是遵循自然规律实现精确的光照模型（lighting model）。

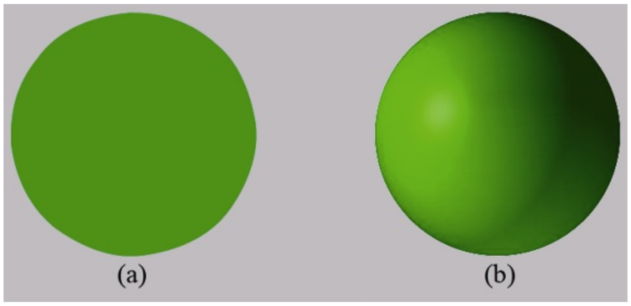


图8.1　球体在有无光照情形下的比较

（a）不受光照的球体看起来像是2D的圆形

（b）受光照射的球体看上去具有立体感

当然，一般来讲，模型越精确则相应的计算代价也就越高昂。这就需要我们在效果的真实性与执行速度上进行权衡。例如，对于电影的3D特效场景而言，可以采用比游戏更复杂、更真实的光照模型，这是因为电影的每一帧都有预渲染处理流程，所以工作者可以用几小时乃至

几天的时间来处理1帧画面。相对来讲，游戏是一种实时（real-time）应用程序，因此每秒至少有30帧的画面需要绘制。

注意，本书对光照模型的讲解与实现，大部分是基于[Möller08]中的描述。

学习目标：

1．对光照与材质的交互有基本的理解。

2．了解局部光照与全局光照之间的差异。

3．探究如何用数学来描述位于物体表面上某点的“朝向”，以此来确定入射光照射到表面的角度。

4．学习如何正确地变换法向量。

5．能够区分环境光、漫反射光以及镜面光。

6．学习如何实现平行光、点光以及聚光灯这3种光源。

7．理解如何通过控制距离函数的衰减参数来实现各种不同的光照强度。

8.1 光照与材质的交互

在开启光照的同时，我们不再直接指出顶点的颜色，而是指定材质与光照，再运用光照方程（lighting equation）基于两者的交互来计算顶点颜色。这样做会使物体的颜色更趋于真实（可再次比较图8.1a与图8.1b之间的天壤之别）。

我们可以把材质看作是确定光照与物体表面如何进行交互的属性集。此属性集中的属性有：表面反射光和吸收光的颜色、表面下材质的折射率、表面的光滑度以及表面的透明度。通过指定材质属性，我们就能为真实世界中如木材、石头、玻璃、金属以及水等不同种类的物质表面进行建模。

在我们所建的模型中，光源可以发出各种强度（intensity）的红、绿、蓝三色混合光。通过这种方式，我们就能模拟出多种多样的光源颜色。当光线从光源发出向外传播并触碰到某个物体时，一部分光线可能会被吸收，而另一部分光线则将被反射（对于玻璃这类透明的物体来说，部分光线能够透过介质（medium，也作媒质），但我们暂不考虑这种情况）。被反射的光线会沿着它的新路径传播，并有可能再次碰到其他物体，发生二次吸收与反射。因此，还会发生这样一种情况，那就是在光线在碰到足够多的物体后被完全吸收了。而一些剩余的光线最终可能会传入观察者的眼中（见图8.2）并照射到视网膜上的感光细胞（视锥细胞与视杆细胞的统称）。

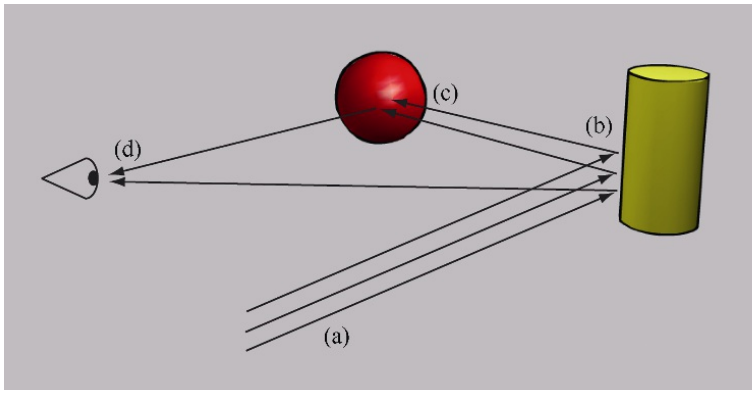


图8.2　（a）白色的入射光　（b）光线触碰到圆柱体，一部分被吸收，其他部分则散射到观察者的眼中或球体上　（c）从圆柱体反射出的光线或被球体吸收，或被球体再次反射，或直接传播至观察者的眼中　（d）观察者眼中所接收到的入射光决定了他所看到的内容

根据三色论（trichromatic theory，又称三色说等，参见[Santrock03]），视网膜含有3种光受体，分别对于红、绿和蓝三色光敏感（有部分重叠）。RGB入射光刺激其相应的光受体，基于光线的强度而产生不同程度的刺激。光受体受到刺激（或没有受到刺激）将神经冲动沿着视神经传至大脑，而大脑再根据光受体所受到的刺激在观察者的头脑中生成相应的画面（当然，如果我们合眼或将眼部遮盖，则受体细胞便不会受到刺激，大脑此时辨识的只有黑色）。

举个例子吧，再次观察图8.2并思考这样一种情况：假设光源发出的是纯白色的光。圆柱体反射75%的红光、75%的绿光，并吸收了其余的光。而球体反射了25%的红光，并吸收了其余的光。由于光线照射到圆柱体时，所有的蓝光被吸收，只有75%的红光和绿光被反射出去（即中高强度的黄色光线）。这些光将发生散射（scatter）现象——一部分射入观察者眼内，一部分朝着球体传播。射入眼中的那部分光线主要刺激红色与绿色的视锥细胞使之较为兴奋，因此观察者会看到呈半光亮

（semi-bright）黄色的圆柱体。就在此时，其他部分光线将继续传播到球体并与之触碰。球体反射25%的红光并吸收其余光线，因此，这些强度本已变弱的入射红光（中高强度红色）将被进一步减弱并反射，而所有的入射绿光则皆被吸收。这些剩余的红光会射入观察者眼中，主要刺激红色视锥细胞，由于刺激不强，产生的兴奋程度并不高，因此观察者会看到着色为暗红色的球体。

在本书中（以及大多数实时应用程序）所采用的光照模型均为局部光照模型（local illumination model）。若使用这种局部模型，则每个物体的光照皆独立于其他物体，我们也就可以在处理光照的过程中仅考虑光源直接发出的光线（即在处理当前的物体光照时，忽略来自场景中其他物体所反弹来的光）。图8.3展示了此模型的效果。

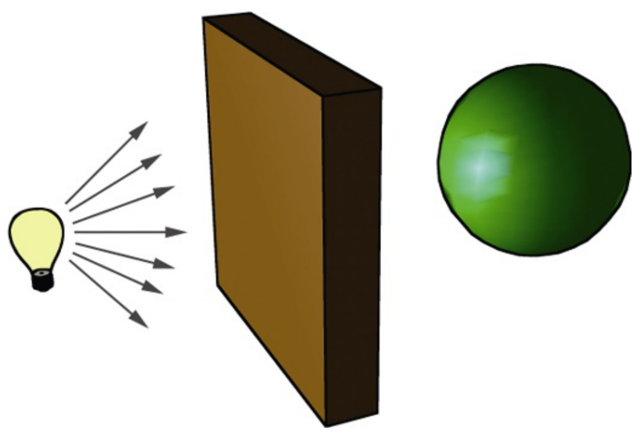


图8.3　从正常的物理角度来说，图中的这堵墙挡住了电灯泡所发出光线的去路，而球体则会生活在墙壁的阴影之下。但是在局部光照模型中，这个球体却受到了光的照射，似乎这堵墙就不曾存在过

谈及全局光照模型（global illumination model），除了要考虑由光源直接发出的光，还要顾及场景中其他物体所反弹来的间接光照。之所

以称为全局光照模型，是因为在对一个物体进行照明时，还要考虑全局场景中的所有事物。全局光照模型的开销通常是实时游戏所负担不起的（但是这种模型会生成十分接近于照片级真实感的场景）。接近于全局光照的实时方法尚处于研究阶段。例如，立体像素全局光照（voxel global illumination）技术（可参见文献《Practical Real-Time Voxel-Based Global Illumination for Current GPUs》）。另一种流行的方法是预计算静态物体（如墙壁、塑像）的间接光照，再用得到的结果来近似地模拟动态物体（如可运动游戏角色）的间接光照。

8.2　法向量

平面法线（face normal，由于在计算机几何学中法线是有方向的向量，所以也有将normal译作法向量）是一种描述多边形朝向（即正交于多边形上所有点）的单位向量，如图8.4a所示。曲面法线（surface normal）是一种垂直于曲面上一点处切平面（有文献强调还要满足曲面法线经过该点这一条件）的单位向量，如图8.4b所示。根据曲面法线即可确定对应曲面上某点的“朝向”。

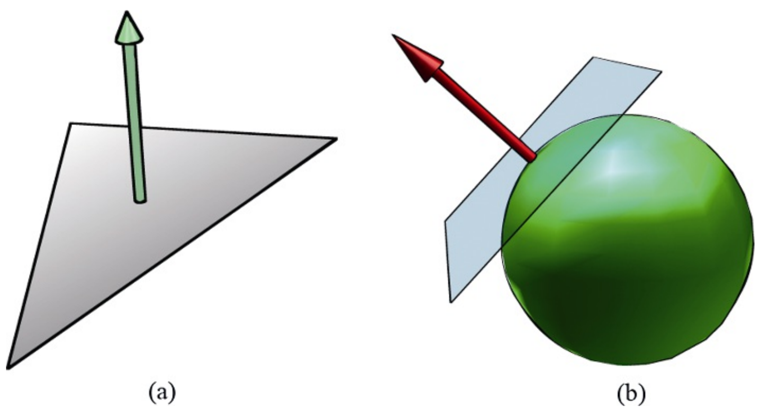


图8.4　平面法线和曲面法线

（a）平面法线正交于对应平面上的所有点　（b）曲面法线是垂直于曲面上某点处切平面的单位向量

对于光照计算来说，我们需要通过三角形网格曲面上每一点处的曲面法线来确定光线照到对应点上的角度。为了求出曲面法线，我们仅先指定位于网格顶点处的曲面法线（所以也将之称作顶点法线，vertex normal）。接下来，为取得三角形网格曲面上每个点处的近似曲面法线，在三角形进行光栅化的过程中对这些顶点法线进行插值计算（参见

图8.5，并回顾5.10.3节）。

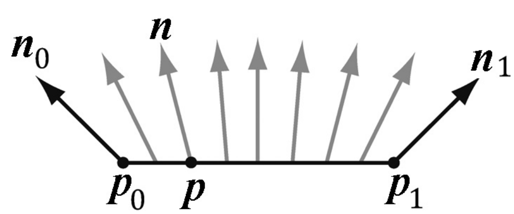


图8.5　$\boldsymbol{n}_0$与$\boldsymbol{n}_1$分别是定义在线段端点$\boldsymbol{p}_0$、$\boldsymbol{p}_1$处的顶点法线。$\boldsymbol{n}$是经此线段端点向量插值（加权平均值）所得到的点$\boldsymbol{p}$处的法向量。即$\boldsymbol{n} = \boldsymbol{n}_0 + t(\boldsymbol{n}_1 - \boldsymbol{n}_0)$，这里的$t$满足$\boldsymbol{p} = \boldsymbol{p}_0 + t(\boldsymbol{p}_1 - \boldsymbol{p}_0)$。尽管我们出于简单而仅介绍了线段的法线插值，但这个方法可以直接推广到3D三角形的情景之中

注意　Note

我们把对每个像素逐一进行法线插值并执行光照计算的方法称为逐像素光照（per pixel lighting）或**phong**光照模型（phong lighting）。而前文所介绍的则是针对每个顶点逐一进行光照计算的逐顶点光照（per vertex lighting）模型，开销虽低但精度也低。接下来，顶点着色器将输出每个顶点光照的计算结果，此时再对三角形中的像素进行插值。将运算作业从像素着色器移至顶点着色器是一种常见的性能优化手段。在以质量为重但又允许结果存在少许视觉偏差的情况下，这种优化方法极具吸引力。

8.2.1　计算法向量

为了找到三角形$\triangle \mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$的平面法线，我们首先计算位于三角形边上的两个向量：

$$\mathbf{u} = \mathbf{p}_1 - \mathbf{p}_0$$

$$\mathbf{v} = \mathbf{p}_2 - \mathbf{p}_0$$

那么，此三角形的平面法线即为：

$$\mathbf{n} = \frac{\mathbf{u} \times \mathbf{v}}{\|\mathbf{u} \times \mathbf{v}\|}$$

下列函数将根据三角形的3个顶点来计算该三角形正面的（详见5.10.2节）平面法线：

```
XMVECTOR ComputeNormal(FXMVECTOR p0,
          FXMVECTOR p1,
          FXMVECTOR p2)
{
  XMVECTOR u = p1 - p0;
  XMVECTOR v = p2 - p0;

  return XMVector3Normalize(
    XMVector3Cross(u,v));
}
```

对于可微的光滑曲面而言，我们可以利用微积分方面的知识来求出曲面点处的法线。但问题在于，三角形网格运用一种被称为求顶点法线平均值（vertex normal averaging）的计算方法。此方法通过对网格中共享顶点$\mathbf{v}$的多边形的平面法线求取平均值，从而获得网格中任意顶点$\mathbf{v}$处的顶点法线$\mathbf{n}$。例如，在图8.6中，网格中的四个多边形共用顶点$\mathbf{v}$，因此，$\mathbf{v}$处的顶点法线求法如下：

$$\mathbf{n}_{\text{avg}} = \frac{\mathbf{n}_0 + \mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3}{\|\mathbf{n}_0 + \mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3\|}$$

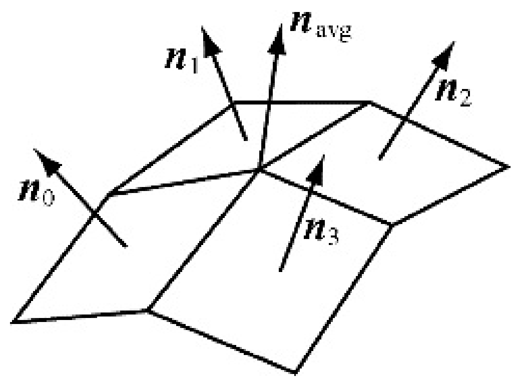


图8.6 位于中间的顶点被四个多边形所共用，所以可通过计算这4个多边形平面法线的平均值来求取中间顶点处的近似法线

在上面这个例子中，由于我们对求和的结果已进行了规范化处理，因此便无需像往常求算术平均值那样再除以4。注意，为了得到更为精准的结果，我们还可以采用更加复杂的求平均值方法，比如说，根据多边形的面积来确定权重（如面积大的多边形的权重要大于面积小的多边形），以求取加权平均值。

下列伪代码展示了若给定三角形网格的顶点列表和索引列表，该如何来求取相应的法线平均值。

```
// 输入
// 1. 一个顶点数组（mVertices）。每个顶点都有一个位置分量（pos）和一个法线分量
（normal）
// 2. 一个索引数组（mIndices）
// 对于网格中的每个三角形来说
for(UINT i = 0; i < mNumTriangles; ++i)
{
  // 第i个三角形的索引
  UINT i0 = mIndices[i*3+0];
  UINT i1 = mIndices[i*3+1];
  UINT i2 = mIndices[i*3+2];

  // 第i个三角形的顶点
  Vertex v0 = mVertices[i0];
  Vertex v1 = mVertices[i1];
  Vertex v2 = mVertices[i2];
```

```
  // 计算平面法线
  Vector3 e0 = v1.pos - v0.pos;
  Vector3 e1 = v2.pos - v0.pos;
  Vector3 faceNormal = Cross(e0, e1);

  // 该三角形共享了下面3个顶点，所以将此平面法线与这些顶点法线相加以求平均值
  mVertices[i0].normal += faceNormal;
  mVertices[i1].normal += faceNormal;
  mVertices[i2].normal += faceNormal;
}

// 对于每个顶点v来说，由于我们已经对所有共享顶点v的三角形的平面法线进行求和，所以现在仅需进行    // 规范化处理即可
for(UINT i = 0; i < mNumVertices; ++i)
  mVertices[i].normal = Normalize(&mVertices[i].normal));
```

8.2.2　变换法向量

思考图8.7a。图中，切向量（tangent vector）$\boldsymbol{u}=\boldsymbol{v}_1-\boldsymbol{v}_0$正交于法向量（normal vector）$\boldsymbol{n}$。如果对此应用一个非等比缩放变换$\boldsymbol{A}$，则可从图8.7b看到，变换后的切向量$\boldsymbol{uA}=\boldsymbol{v}_1\boldsymbol{A}-\boldsymbol{v}_0\boldsymbol{A}$没能与变换后的法向量$\boldsymbol{nA}$继续保持正交性。

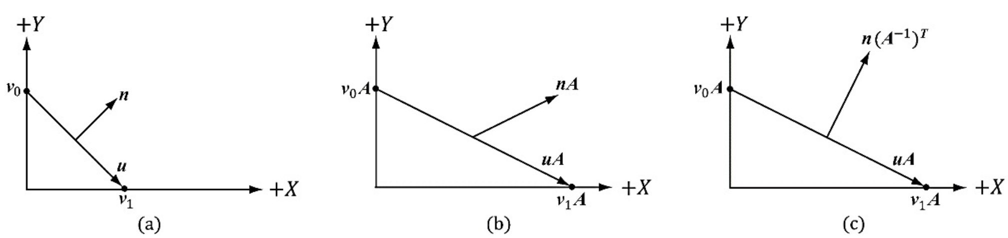


图8.7　切向量与法向量的变换

（a）切向量与法向量在变换前的正交关系

（b）经过在x轴正方向放大两倍的处理后，法向量不再与切向量保持正交关系

（c）通过对法向量进行图b中缩放变换的逆转置矩阵运算后，法向量与切向量重归正交关系

所以，我们现在所面对的问题是，若给定一个用于变换点与向量

（非法线）的变换矩阵$\boldsymbol{A}$，如何能够求出这样一个变换矩阵$\boldsymbol{B}$：通过它来变换法向量，使经矩阵$\boldsymbol{A}$变换后的切向量与法向量重归正交的关系（即$\boldsymbol{uA}\cdot\boldsymbol{nB}=0$）。为此，我们首先从已知的信息着手，如果法向量$\boldsymbol{n}$正交于切向量$\boldsymbol{u}$，则有：

$\boldsymbol{u}\cdot\boldsymbol{n}=0$	切向量正交于法向量
$\boldsymbol{un}^{\mathrm{T}}=0$	将点积改写为矩阵乘法
$\boldsymbol{u}(\boldsymbol{AA}^{-1})\boldsymbol{n}^{\mathrm{T}}=0$	插入单位矩阵$\boldsymbol{I}=\boldsymbol{AA}^{-1}$
$(\boldsymbol{uA})(\boldsymbol{A}^{-1}\boldsymbol{n}^{\mathrm{T}})=0$	根据矩阵乘法运算的结合律
$(\boldsymbol{uA})((\boldsymbol{A}^{-1}\boldsymbol{n}^{\mathrm{T}})^{\mathrm{T}})^{\mathrm{T}}=0$	根据转置矩阵的性质$(\boldsymbol{A}^{\mathrm{T}})^{\mathrm{T}}=\boldsymbol{A}$
$(\boldsymbol{uA})(\boldsymbol{n}(\boldsymbol{A}^{-1})^{\mathrm{T}})^{\mathrm{T}}=0$	根据转置矩阵的性质$(\boldsymbol{AB})^{\mathrm{T}}=\boldsymbol{B}^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}$
$\boldsymbol{uA}\cdot\boldsymbol{n}(\boldsymbol{A}^{-1})^{\mathrm{T}}=0$	将矩阵乘法改写为点积的形式
$\boldsymbol{uA}\cdot\boldsymbol{nB}=0$	变换后的切向量正交于变换后的法向量

因此，通过$\boldsymbol{B}=(\boldsymbol{A}^{-1})^{\mathrm{T}}$（矩阵$\boldsymbol{A}$的逆转置矩阵）对法向量进行变换后，即可使它垂直于经矩阵$\boldsymbol{A}$变换后的切向量$\boldsymbol{uA}$。

注意，如果矩阵$\boldsymbol{A}$是正交矩阵（即满足$\boldsymbol{A}^{\mathrm{T}}=\boldsymbol{A}^{-1}$），那么$\boldsymbol{B}=(\boldsymbol{A}^{-1})^{\mathrm{T}}=(\boldsymbol{A}^{\mathrm{T}})^{\mathrm{T}}\boldsymbol{A}$。也就是说，在这种情况下我们无需再计算它的逆转置矩阵，因为利用正交矩阵$\boldsymbol{A}$自身即可实现这一变换。总而言之，当我们需要对经过非等比变换或剪切变换（shear transformation，也有译作切变转变等）后的法向量进行变换时，则可使用逆转置矩阵。

我们在头文件MathHelper.h中为计算逆转置矩阵实现了一个辅助函数：

```
static XMMATRIX InverseTranspose(CXMMATRIX M)
{
  XMMATRIX A = M;
  A.r[3] = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);

  XMVECTOR det = XMMatrixDeterminant(A);
  return XMMatrixTranspose(XMMatrixInverse(&det, A));
}
```

在通过逆转置矩阵对向量进行变换时，我们可以将向量变换矩阵中与平移操作有关的项清零，而只允许点类才有平移变换。然而，从3.2.1节中可知，（在使用齐次坐标的情况下）将向量的第4个分量设置为$w=0$，就可以防止向量因平移操作而受到影响。从这个角度来讲，我们便无须为矩阵中的平移项置零。但问题在于，如果我们希望连接逆转置矩阵以及另一个不含非等比缩放的矩阵，如观察矩阵$(\boldsymbol{A}^{-1})^{\mathrm{T}}\boldsymbol{V}$，那么，$(\boldsymbol{A}^{-1})^{\mathrm{T}}$中经转置后的第4列平移项将“渗入”最终的乘积矩阵，从而导致错误的计算结果。就此而言，我们对矩阵中的平移项置零是避免这个错误的预防措施。而变换法线所采用的正确公式实则为$((\boldsymbol{AV})^{-1})^{\mathrm{T}}$。下面是一个缩放与平移矩阵的示例，可以看出，经过逆转置变换后，矩阵的第4列并不是$[0,0,0,1]^{\mathrm{T}}$：

$$\boldsymbol{A}=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0.5 & 0 & 0\\ 0 & 0 & 0.5 & 0\\ 1 & 1 & 1 & 1\end{bmatrix}$$

$$(\boldsymbol{A}^{-1})^{\mathrm{T}}=\begin{bmatrix}1 & 0 & 0 & -1\\ 0 & 2 & 0 & -2\\ 0 & 0 & 2 & -2\\ 0 & 0 & 0 & 1\end{bmatrix}$$

注意 Note

尽管运用了逆转置变换，但法向量仍可能会失去其单位长度。所以在变换完成后，可能需对它再次进行规范化处理。

8.3　参与光照计算的一些关键向量

在本节中，我们要介绍一些在光照计算中起重要作用的向量。如图8.8所示，$\boldsymbol{E}$是观察者的观察位置，我们现在来考察：在观察点$\boldsymbol{p}$处沿着单位向量$\boldsymbol{v}$所定义的视线来进行观察的过程。位于表面的点$\boldsymbol{p}$处有法线$\boldsymbol{n}$，光线由入射方向$\boldsymbol{I}$照射到点$\boldsymbol{p}$。光向量（light vector）$\boldsymbol{L}$为单位向量，其所指方向与照射到表面上点$\boldsymbol{p}$处入射光线$\boldsymbol{I}$的方向刚好相反。尽管在工作中使用光的入射方向$\boldsymbol{I}$可能更为直观，但是为了进行光照计算，我们还是采用光向量$\boldsymbol{L}$。对于朗伯余弦定律（详见8.4小节）而言，向量$\boldsymbol{L}$用于计算$\boldsymbol{L}\cdot\boldsymbol{n}=\cos\boldsymbol{\theta}_i$，其中的$\cos\boldsymbol{\theta}_i$是光向量$\boldsymbol{L}$与法向量$\boldsymbol{n}$之间的夹角。反射向量$\boldsymbol{r}$是入射光向量$\boldsymbol{L}$关于表面法线$\boldsymbol{n}$的镜像。观察向量（view vector，或to-eye vector）$\boldsymbol{v}=\mathrm{normalize}\boldsymbol{E}-\boldsymbol{p}$）[1]是从表面上的点$\boldsymbol{p}$到观察点$\boldsymbol{E}$方向上的单位向量，它定义了由观察点向表面点观察的视线。我们有时还要用到向量$-\boldsymbol{v}$，它是我们所要计算的由观察点到表面点这条光线路径上的单位向量。

反射向量的定义为$\boldsymbol{r}=\boldsymbol{I}-2(\boldsymbol{n}\cdot\boldsymbol{I})\boldsymbol{n}$，如图8.9所示（这里假设$\boldsymbol{n}$为单位向量）。然而，我们在着色器中实际上是利用内置函数`reflect`来计算$\boldsymbol{r}$的。

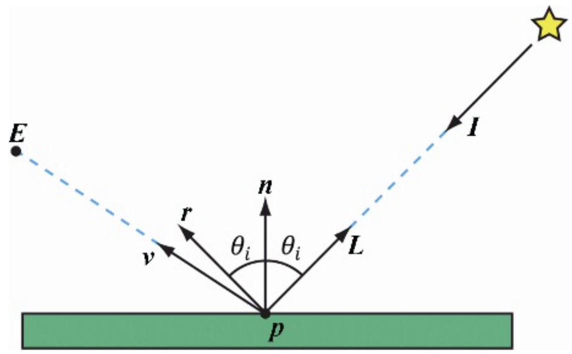


图8.8　光照计算中所需要的各种重要向量

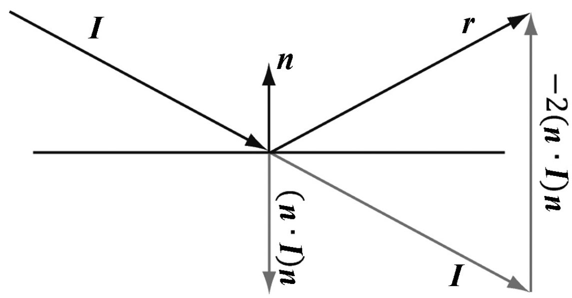


图8.9　光线的反射示意图

8.4　朗伯余弦定律

我们可以将光看作是光子的集合，在空间中按特定的方向传播。每个光子都载有（光）能量。光源每秒发出的（光）能量称为辐射通量（radiant flux）。而单位面积上的辐射通量密度（irradiance，称为辐（射）照度）是一种很重要的概念，因为我们将用它来确定表面某区域所接收到的光量（即眼睛感受到的明亮度）。一般来讲，我们可以认为辐照度是照射到表面某区域的光量，或者是通过空间中某假想区域的光量。

光线垂直照射到表面（即光向量$\boldsymbol{L}$与法向量$\boldsymbol{n}$相等时）的强度要大于以某个角度照射到表面的强度。试想有一小束辐射通量为P且横截面面积为A_1的光束。如果将此光束正向垂直打向表面（见图8.10a），则光束照射到表面上的面积为A_1，而A_1内的辐照度为$E_1 = P/A_1$。现假设转动光源，使光束以某个入射角度照射到表面上（如图8.10b），则光束将覆于表面上的更大面积A_2。此时，该面积的辐照度为$E_2 = P/A_2$。根据三角学可知，A_1及A_2的关系为：

$$\cos\boldsymbol{\theta} = \frac{A_1}{A_2} \Rightarrow \frac{1}{A_2} = \frac{\cos\boldsymbol{\theta}}{A_1}$$

所以，

$$E_2 = \frac{p}{A_2} = \frac{p}{A_1}\cos\boldsymbol{\theta} = E_1\cos\boldsymbol{\theta} = E_1\left(n \cdot L\right)$$

换句话说，面积A_2内的辐照度就相当于将受垂直方向光照的面积A_1

内的辐照度按比例$\boldsymbol{n} \cdot \boldsymbol{L} = \cos\boldsymbol{\theta}$进行缩放。这就是传说中的朗伯余弦定律（Lambert’s Cosine Law）。考虑到光线照射到表面另一侧的情况（此时，点积的结果为负值），我们用max函数来钳制[2]“缩放因子”的取值范围：

$$f(\boldsymbol{\theta}) = \max(\cos\boldsymbol{\theta},\ 0) = \max(\boldsymbol{L} \cdot \boldsymbol{n},\ 0)$$

图8.11所示的是函数$f(\boldsymbol{\theta})$的图像。观察可知，随着变量$\boldsymbol{\theta}$的变化，函数的值域范围为0.0～1.0（即光照强度的变化范围为0%～100%）。

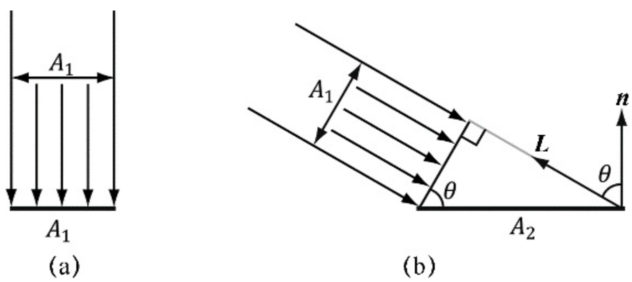


图8.10　照射方向的对比

（a）横截面面积为A_1的光束垂直照射到表面　（b）将横截面面积为A_1的光束以某角度照射到表面上的最大面积为A_2，因此等量光能将扩散到更大的面积上，继而导致照射到物体表面的光束看起来“偏暗”

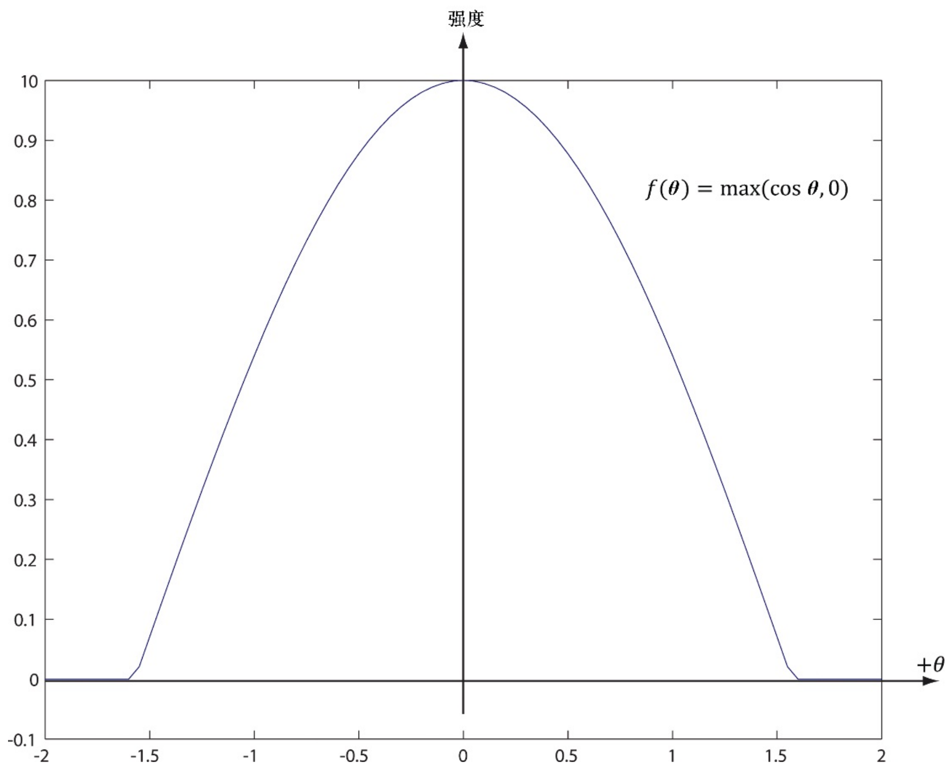


图8.11　在范围$-2 < \boldsymbol{\theta} < 2$内，函数$f(\boldsymbol{\theta}) = \max(\cos\boldsymbol{\theta},\ 0) = \max(L \cdot n,\ 0)$的图像。注意，$\pi/2 \approx 1.57$

8.5　漫反射光照

考虑图8.12所示的不透明物体的表面。当光线照射到表面上的某一点时，一部分光会进入物体的内部，并与表面附近的物质相互作用。这些光会在物体内部四处反弹，其中一部分会被吸收，而余下部分则会向各个方向散射并返回表面，这即是所谓的漫反射（diffuse reflection）。为了方便，我们假设光在入射点处发生散射。光的吸收和散射程度与物体的材质密切相关，例如，木材、泥土、砖块、瓦片与灰泥所吸收与散射的光量是不同的（这也正是为什么不同材质看起来各不相同的原因）。在所用的这种光照与材质交互的近似模型之中，我们规定光线会在表面的所有方向上均匀散射，因此，无论在哪个观察点（眼睛观看的位置）进行观察，反射光都会进入观察者的眼中。所以，我们无须考虑观察点的具体位置（即漫反射光照的计算与观察点无关），而表面点上的颜色在任何观察点上看来也都是相同的。

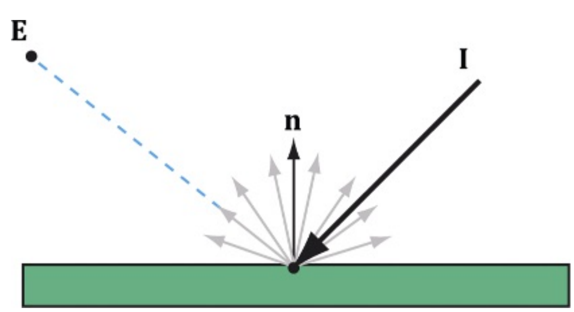


图8.12　当入射光照射到一个漫反射面时便会向各个方向均匀散射。光线进入介质的内部后，便会在其表面下发生散射。一部分光将被物体吸收，而剩余部分则会散射回物体的表面。由于表面下的散射模型过于复杂，我们便假设光在入射点处均匀的散射向表面外的各个方向

我们将漫反射光照的计算分为两个部分。在第一部分中，我们要指

定光照颜色以及漫反射反照率（diffuse albedo）颜色。漫反射反照率表示的是根据表面的漫反射率（diffuse reflectance，也译作漫反射比）而被反射的入射光量（根据能量守恒定律，入射光不是被反射回表面就是被材质吸收了）[3]。要对它们进行处理就需用分量式颜色乘法（因为光是具有颜色的）。例如，假设表面上一点会反射50%的入射红光、100%的绿光以及75%的蓝光，如果这时有一束强度为80%的白光袭来，那么此入射光的量值就可以表示为$\boldsymbol{B}_L=(0.8,0.8,0.8)$，又因为漫反射反照率为$\boldsymbol{m}_d=(0,5,1.0,0.75)$，所以此点的反射光量为：

$$\boldsymbol{c}_{\mathrm{d}}=\boldsymbol{B}_{\mathrm{L}}\otimes\boldsymbol{m}_{\mathrm{d}}=(0.8,\ 0.8,\ 0.8)\otimes(0.5,\ 1.0,\ 0.75)=(0.4,\ 0.8,\ 0.6)$$

可以发现，漫反射反照率分量的取值范围必定在0.0～1.0之间，因此我们用小数来表示反射光。

然而，上述公式并非十分准确，我们还需将朗伯余弦定律（根据表面法线与光向量之间的夹角来控制表面接收原始光照的量）考虑在内。设$\boldsymbol{B}_L$表示入射光量，$\boldsymbol{m}_d$为漫反射反照率，$\boldsymbol{L}$为光向量，而$\boldsymbol{n}$为表面法线，则位于表面上某点处的漫反射光量为：

$$\boldsymbol{c}_{\mathrm{d}}=\max(\boldsymbol{L}\cdot\boldsymbol{n},\ 0)\cdot\boldsymbol{B}_{\mathrm{L}}\otimes\boldsymbol{m}_{\mathrm{d}}\qquad（8.1）$$

8.6 环境光照

如前文所述，我们的光照模型并没有考虑到场景中其他物体反射来的间接光照。事实上，我们在真实世界中所看到的多是间接光。例如，屋内的光源并没有路径可以直接照入连接房间的走廊，但是光线经过室内墙壁的反射并射入走廊，使它有了些许光亮。再来看另一个示例，假设我们正坐在一间设有一个光源的屋子里，房内的桌上还放有一个茶壶。虽然茶壶只有一侧被光源直接照射，但它的另一侧却不可能完全被笼罩在漆黑的阴影当中。这是因为一部分光经墙壁或室内的其他物体反射，而最终照射到茶壶的背面。

为了处理这种间接光照，我们给光照方程引进了一个环境光（ambient light）项：

$$\boldsymbol{c}_{\mathrm{a}} = \boldsymbol{A}_{\mathrm{L}} \otimes \boldsymbol{m}_{\mathrm{d}} \qquad (8.2)$$

颜色$\boldsymbol{A}_L$指定了表面收到的间接（环境）光量，它可能与光源发出的光量不同，因为光源发射的光在其他表面反射的时候会被吸收一部分。漫反射反照率$\boldsymbol{m}_d$指示了根据表面漫反射率而被表面反射的入射光量。同时，我们也借用此值来表明被表面反射的入射环境光量。也就是说，对于环境光照而言，我们其实是在围绕间接（环境）光照的漫反射率进行建模。所有的环境光都是以统一的亮度将物体稍稍照亮——而完全没有按真实的物理效果进行计算。这个模型的总体思路是：间接光照在场景中会发生多次的散射与反射，并会在所有方向上均等的射向目标

物体。

8.7　镜面光照

我们此前用漫反射光照来模拟漫反射的过程：光进入介质，发生反射，部分光被吸收，而剩下的光则向介质外的各个方向散射。第二种反射的发生是根据一种名为菲涅耳效应（Fresnel effect，也译作菲涅尔效应）的物理现象。当光线到达两种不同折射率（index of refraction）介质之间的界面时，一部分光将被反射，而剩下的光则发生折射（refract），这个过程可参考图8.13。折射率是一种介质的物理性质，即光在真空中传播的速度与光在给定介质内的传播速度之比。我们将第二种光的反射过程称为镜面反射（specular reflection），把被反射的光称为镜面（反射）光（specular light），如图8.14a所示。

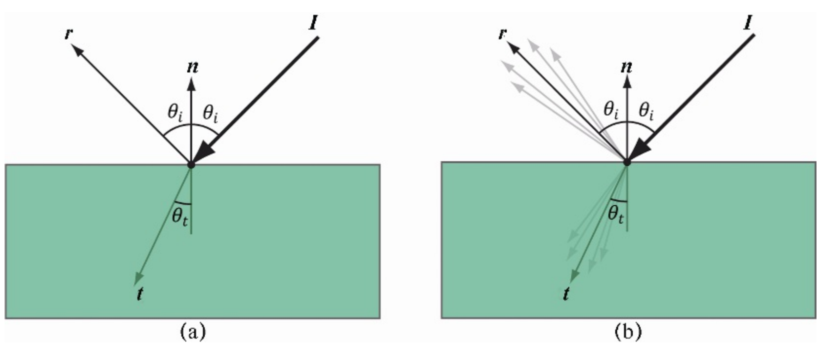


图8.13　镜面光照

（a）具有法线$\boldsymbol{n}$的完全光滑平整的镜面（即理想镜面）所呈现的菲涅耳效应。入射光$\boldsymbol{I}$抵达表面后分为两个部分进行传播，一部分按反射方向$\boldsymbol{r}$发生反射，剩余部分则以折射方向$\boldsymbol{t}$折射入介质。所有这些向量都位于同一平面内。反射向量$\boldsymbol{r}$与法线$\boldsymbol{n}$之间的夹角总保持为$\boldsymbol{\theta}_i$，而且光向量$\boldsymbol{L}=-\boldsymbol{I}$与法线$\boldsymbol{n}$之间的夹角也同为$\boldsymbol{\theta}_i$。至于折射向量$\boldsymbol{t}$与$-\boldsymbol{n}$之间的夹角$\boldsymbol{\theta}_t$，则取决于两种介质间的折射率以及斯涅尔折射定律（Snell’s Law）　（b）事实上，大多数的物体并不是完全光滑平整的理想镜面，而是在微观上具有一定的粗糙度。这就导致了反射光与折射光分别关于反射向量与折射向量产生一些扩散

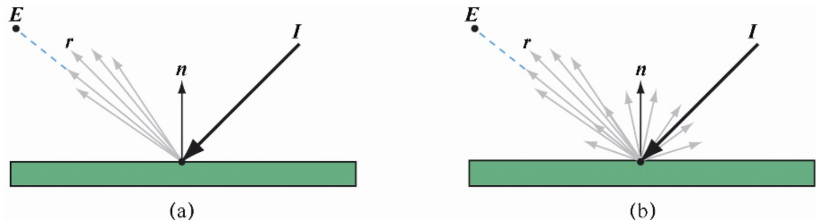


图8.14　粗糙表面的镜面光照

（a）粗糙表面的镜面光会在向量$\boldsymbol{r}$的附近发生些许扩散

（b）射入观察者眼中的反射光实为镜面反射和漫反射的组合效果

如果折射光沿折射向量从介质的另一侧射出，并进入观察者的眼中，则该物体看起来就像是透明的。即光通过了透明的物体。在实时的图像处理中，一般用alpha混合技术或后期处理特效来模拟透明对象的折射过程，有关内容在本书后面会有相应的讲解。现在，我们只考虑不透明的物体。

对于不透明物体而言，折射光进入介质并根据漫反射率发生漫反射。所以通过对图8.14b中不透明物体的观察可以看出，从表面反射和进入眼睛的光量是由物体的（漫）反射光和镜面光所构成的。与漫反射光相比，镜面光可能并不会射入观察者的眼内，因为其反射只发生在某一特定角度。也就是说，镜面光照的计算与观察点有关。同时，这就意味着随着观察位置在场景中的移动，它所收到的镜面光量也将随之发生改变。

8.7.1　菲涅耳效应

我们来考虑一个具有法线$\boldsymbol{n}$的平滑界面，它将两种不同折射率的介质分隔开来。由于在界面处具有折射率不连续性（因不同介质的折射率

差异所导致），当光线照射到界面时，一部分会被界面反射，另一部分则折射进界面（见图8.13）。菲涅耳方程（Fresnel equations）以数学方法描述了入射光线被反射的百分比，即$0 \leqslant \boldsymbol{R}_F \leqslant 1$。根据能量守恒定律，如果$\boldsymbol{R}_F$是反射光量，则$(1-\boldsymbol{R}_F)$为折射光量。$\boldsymbol{R}_F$的值是一个RGB向量，因为光的颜色反映了反射光量。

反射的光量既依赖于介质（某些材质的反射率相对更大），也与法向量$\boldsymbol{n}$与光向量$\boldsymbol{L}$之间的夹角$\boldsymbol{\theta}_i$有关。由于光照过程的复杂性，我们一般不会将完整的菲涅耳方程用于实时渲染，而是采用石里克近似（Schlick approximation）法来加以代替：

$$\boldsymbol{R}_F(\boldsymbol{\theta}_i) = \boldsymbol{R}_F(0°) + (1 - \boldsymbol{R}_F(0°))(1 - \cos\boldsymbol{\theta}_i)^5$$

$\boldsymbol{R}_F(0°)$是介质的一种属性，下面所列的即是一些常见材质的对应属性数值[Möller08]。

介质	$\boldsymbol{R}_F(0°)$
水	(0.02, 0.02, 0.02)
玻璃	(0.08, 0.08, 0.08)
塑料	(0.05, 0.05, 0.05)
金	(1.0, 0.71, 0.29)

银	(0.95, 0.93, 0.88)
铜	(0.95, 0.64, 0.54)

图8.15所示的是3种具有不同$\boldsymbol{R}_F(0°)$属性值材质的石里克近似曲线图。此图的观察要点在于反射光量随着$\boldsymbol{\theta}_i \to 90°$增加而递增的过程。我们来看一个现实世界中的例子，思考图8.16，现假设我们正置身于一个水质相对清澈、深达数英尺的小池塘里。若向下俯视，我们基本可以清楚地看到其底部沉积的沙石。这是由于有从周围环境中照射到池水的光以接近于$0.0°$的小角度θ_i反射进我们的眼睛而造成的；如此一来，反射到我们眼中的光量相对较低，又根据能量守恒定律可知，此时的折射光量却很高。现在来换个观察角度，如果我们向稍远处望去，将会看到池水极强的反射光。这是因为有从周围环境中射向水的光以接近$90.0°$的角度θ_i反射进我们的眼中，从而增加了反射光量。这种现象通常称为菲涅耳效应（Fresnel effect）。可以将菲涅耳效应简洁地概括为：反射光量取决于材质（$\boldsymbol{R}_F(0°)$）以及法线与光向量之间的夹角。

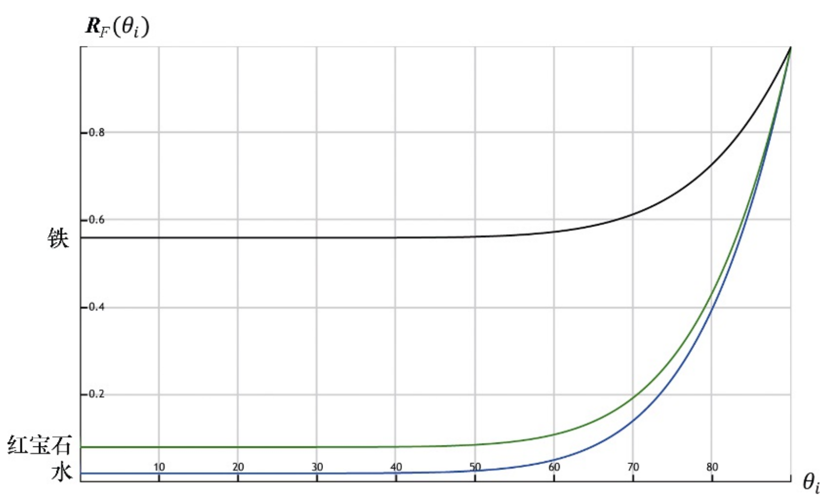


图8.15　水、红宝石、铁3种不同材质的石里克近似曲线图

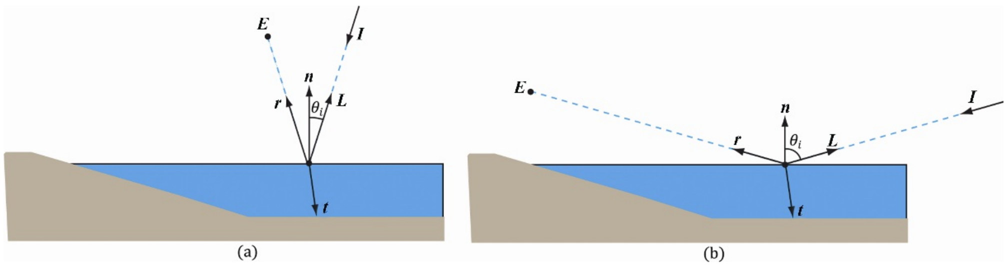


图8.16　池面的反射与折射

（a）俯视池底的时候，由于光向量$\boldsymbol{L}$与法线$\boldsymbol{n}$之间的夹角极小，因此反射光量低而折射光量高

（b）从远处向池水瞭望时，因为光向量$\boldsymbol{L}$与法线$\boldsymbol{n}$之间的夹角过大，从而导致反射光量高而折射光量低

金属会吸收透射光（transmitted light）[Möller08]，这意味着它们不具有本体反射率（body reflectance，也有译作体反射，或体漫反射等）。虽然如此，但金属并不会看上去表现为纯黑色，因为它们的$\boldsymbol{R}_F(0°)$值较高，也就是说，就算是在接近于$0°$这样的极小入射角度上，它们也能反射可观的镜面光量。

8.7.2　表面粗糙度

真实世界中的反射物体往往不是理想镜面（perfect mirror）。尽管一个物体的表面看起来似乎十分平滑，但从微观水平上看，它还是具有一定的粗糙度（roughness）。如图8.17所示，我们可以认为理想镜面的粗糙度为0，它的微观表面法线（micro-normal，或作微表面法线）都与宏观表面法线（macro-normal，或作宏表面法线）的方向相同。随着粗糙度的增加，微观表面法线的方向开始纷纷偏离宏观表面法线，由此反

射光逐渐扩展为一个镜面瓣（specular lobe）。

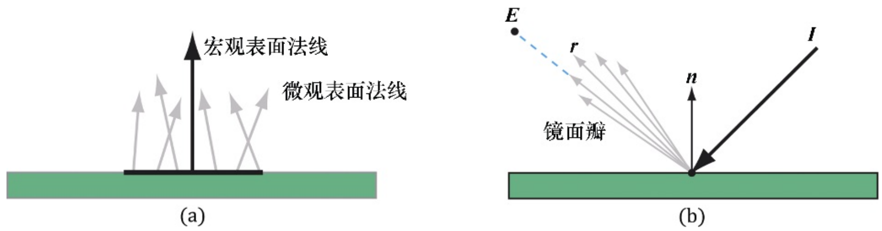


图8.17　图a中，黑色的水平线条表示被放大的小面元（small surface element）。从微观角度来看，由于在此层级的表面上具有一定的粗糙度，因此许多微观表面法线各指向不同的方向。若表面愈平滑，便会有更多的微观表面法线愈发平行于宏观表面法线；而表面越粗糙，则会有更多的微观表面法线越发偏离于宏观表面法线。图b中，粗糙度令镜面反射光扩散开来，镜面反射光的范围称为镜面瓣。一般来讲，镜面瓣的形状将根据建模所用的表面材质种类而各不相同

为了用数学方法对粗糙度进行建模，我们采用了微平面（microfacet，也作微表面）模型。在此模型中，我们将微观表面模拟为由多个既微小又平滑的微平面所构成的集合；而微观表面法线正是这些微平面上的法线。针对指定的观察单位向量$\boldsymbol{v}$以及光向量$\boldsymbol{L}$，我们需要了解由$\boldsymbol{L}$向$\boldsymbol{v}$反射的所有微平面片段的分布情况；换言之，即法线为$\boldsymbol{h}=\text{normalize}(\boldsymbol{L}+\boldsymbol{v})$这种微平面片段在所有微平面中所占比例，如图8.18所示。这样一来，就可以确定有多少光通过镜面反射的方式，以此路径进入到观察者的眼中——发生由$\boldsymbol{L}$到$\boldsymbol{v}$反射过程的微平面越多，则观察者在此角度上看到的镜面光越明亮。

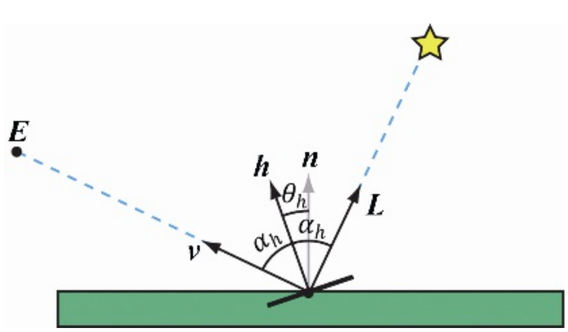


图8.18　以$\boldsymbol{h}$为法线，光从$\boldsymbol{L}$反射至$\boldsymbol{v}$的微平面

由于向量$\boldsymbol{h}$位列向量$\boldsymbol{L}$与向量$\boldsymbol{v}$间的中间位置，故称之为中间向量（halfway vector，也有译作中途向量、半角向量等）。此外，还要再引进一个中间向量$\boldsymbol{h}$与宏观表面法线$\boldsymbol{n}$之间的夹角$\boldsymbol{\theta}_h$。

我们定义归一化分布函数$\rho(\boldsymbol{\theta}_h) \in [0, 1]$，用来表示微观表面法线$\boldsymbol{h}$与宏观表面法线$\boldsymbol{n}$之间夹角为$\boldsymbol{\theta}_h$的微平面的分布情况。从直观上来讲，我们希望函数$\rho(\boldsymbol{\theta}_h)$在$\boldsymbol{\theta}_h = 0°$时取得最大值。也就是说，我们希望微平面法线都平行于宏观表面法线。并盼望随着$\boldsymbol{\theta}_h$的增加（即渐渐偏离于宏观表面法线的微观表面法线$\boldsymbol{h}$），这些以向量$\boldsymbol{h}$为法线的微平面片段逐渐减少。用来模拟以上讨论中期望模型$\rho(\boldsymbol{\theta}_h)$的一种较流行的可控函数为：

$$\begin{aligned}\rho(\boldsymbol{\theta}_h) &= \cos^m(\boldsymbol{\theta}_h) \\ &= \cos^m(\boldsymbol{n} \cdot \boldsymbol{h})\end{aligned}$$

注意到，为了求出$\cos(\boldsymbol{\theta}_h) = (\boldsymbol{n} \cdot \boldsymbol{h})$就需要知道式中这两种向量。图8.19展示了对于取不同m值时函数$\rho(\boldsymbol{\theta}_h) = \cos^m(\boldsymbol{\theta}_h)$的图像。其中，变量$m$控制的是粗糙度，它指定了所有以微观表面法线$\boldsymbol{h}$与宏观表面法线$\boldsymbol{n}$之间夹角为$\boldsymbol{\theta}_h$的微平面片段的分布情况（所占比例）。随着$m$的减小，表面变得更加粗糙，而微平面的法线也都愈加偏离宏观表面法线。随着m的增大，表面变得更加光滑，微平面法线也都越发趋于宏观表面法线。

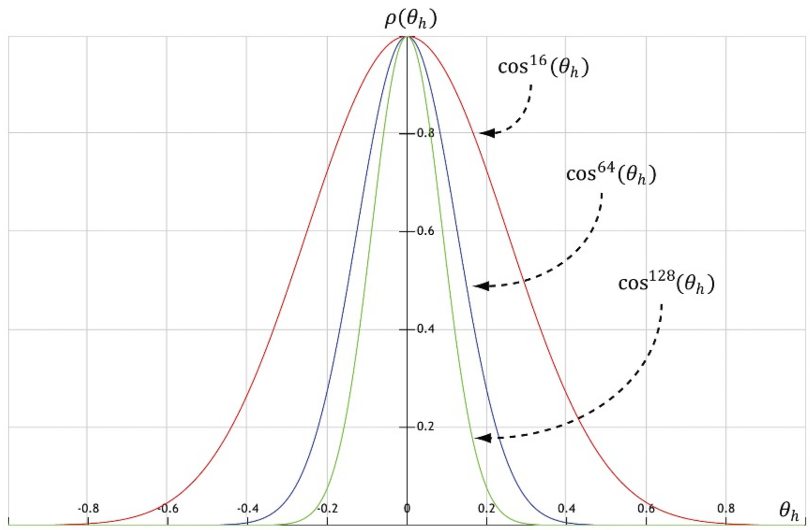


图8.19　一种针对粗糙度进行建模的函数曲线图

我们可以将$\rho(\boldsymbol{\theta}_h)$与某种归一化因子进行组合，从而获得基于粗糙度来模拟镜像反射光量的新函数：

$$
\begin{aligned}
S(\boldsymbol{\theta}_h) &= \frac{m+8}{8}\cos^m(\boldsymbol{\theta}_h) \\
&= \frac{m+8}{8}(\boldsymbol{n}\cdot\boldsymbol{h})^m
\end{aligned}
$$

图8.20展示了此函数根据不同取值的变量m所得到的图像。与之前一样，变量m依然控制着粗糙度，但是我们还为之加上了归一化因子$\frac{m+8}{8}$，以使光能守恒。该归一化因子实际上支配着图8.20中曲线的高度，因此随着变量m的变化使镜面瓣变宽或变窄，令光能整体达到守恒。对于较小的m值来说，表面会更加粗糙，并且镜面瓣变宽，继而促使光能散播得更广。因此，我们预计镜面高光（specular highlight，即镜面光）会变暗，因为其能量已经被广泛播散出去了。另一方面，对于较大的m值而言，表面会更加平滑，而镜面瓣会变得更窄。因此，我们预计镜面高光会更亮，因为其能量更为集中。从几何角度上来看，变量m

控制着镜面瓣的扩散程度。如果要模拟光滑的表面（例如被抛光的金属）就使用较大的m值，而针对更粗糙的表面而言，则使用较小的m值。

我们把菲涅耳反射以及表面粗糙度这两个公式的组合来作为这一节的尾声。首先尝试计算在观察方向$\boldsymbol{v}$上所反射的光量（见图8.18）。回顾以$\boldsymbol{h}$为法线，且反射光沿观察向量$\boldsymbol{v}$传播的微平面。设α_h为光向量$\boldsymbol{v}$与中间向量$\boldsymbol{h}$之间的夹角，那么，根据菲涅耳效应所言，$\boldsymbol{R}_F\left(\alpha_h\right)$反映了关于$\boldsymbol{h}$射入$\boldsymbol{v}$的反射光量。又由于表面的粗糙度为$S\left(\boldsymbol{\theta}_h\right)$，所以我们还要将反射光量$\boldsymbol{R}_F\left(\alpha_h\right)$乘以粗糙度。由此，我们便可得到镜面反射光量：设$(\max(\boldsymbol{L}\cdot\boldsymbol{n},0)\cdot\boldsymbol{B}_{\mathrm{L}})$表示入射光照到表面上一点的光量，而根据粗糙度与菲涅耳效应，$(\max(\boldsymbol{L}\cdot\boldsymbol{n},0)\cdot\boldsymbol{B}_{\mathrm{L}})$这一段镜面反射到观察者眼中的实际光量为：

$$c_{\mathrm{s}}=\max\left(\boldsymbol{L}\cdot\boldsymbol{n},0\right)\cdot\boldsymbol{B}_{\mathrm{L}}\otimes\boldsymbol{R}_F\left(\alpha_h\right)\frac{m+8}{8}(\boldsymbol{n}\cdot\boldsymbol{h})^m \tag{8.3}$$

可以发现，如果$\boldsymbol{L}\cdot\boldsymbol{n}\leq 0$，那么所计算的结果是射向表面另一侧的光，因而此时的正表面并不会接收到任何光照。

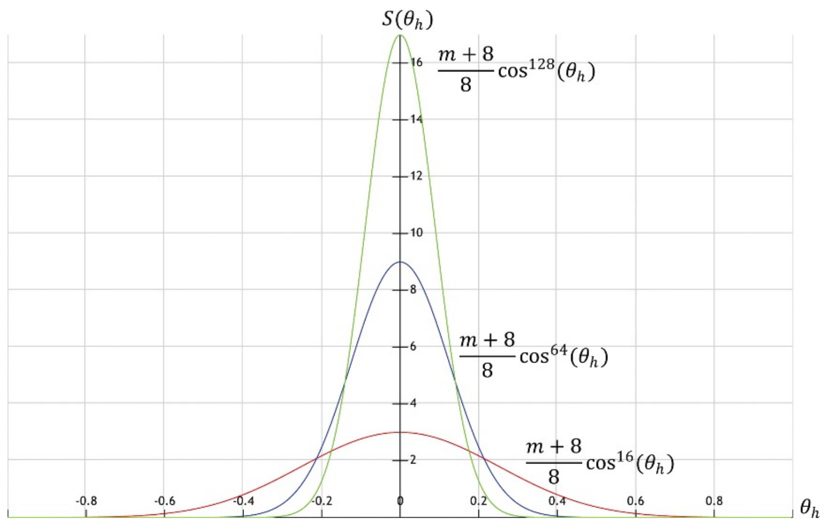


图8.20　一种根据粗糙度来对镜面反射光进行建模的函数图像

8.8　光照模型的概述

现在将之前所述的所有光照内容都结合起来，即表面反射的光量相当于环境反射光、漫反射光以及镜面反射光的光量总和。

1．环境光$\boldsymbol{c}_a$：模拟经表面反射的间接光量。

2．漫反射光$\boldsymbol{c}_d$：对进入介质内部，又经过表面下吸收而最终散射出表面的光进行模拟。由于对表面下的散射光建模比较困难，我们便假设在表面下与介质相互作用后的光从进入表面处返回，并向各个方向均匀散射。

3．镜面光$\boldsymbol{c}_s$：模拟经菲涅耳效应与表面粗糙度共同作用的表面反射光。

据此来推导出我们在本书的着色器中实现的光照方程：

$$\begin{aligned}\text{LitColor} &= \boldsymbol{c}_\text{a} + \boldsymbol{c}_\text{d} + \boldsymbol{c}_\text{s} \\ &= \boldsymbol{A}_\text{L} \otimes \boldsymbol{m}_\text{d} + \max(\boldsymbol{L}\cdot\boldsymbol{n}, 0)\cdot\boldsymbol{B}_\text{L} \otimes \left(\boldsymbol{m}_\text{d} + \boldsymbol{R}_F(\alpha_h)\frac{m+8}{8}(\boldsymbol{n}\cdot\boldsymbol{h})^m\right)\end{aligned}$$

（8.4）

设式（8.4）中的所有向量均为单位长度。

1．$\boldsymbol{L}$：指向光源的光向量。

2．$\boldsymbol{n}$：表面法线。

3．$\boldsymbol{h}$：列于光向量与观察向量（由表面点指向观察点的单位向量）之间的中间向量。

4．$\boldsymbol{A}_L$：表示入射的环境光量。

5．$\boldsymbol{B}_L$：表示入射的直射光量。

6．$\boldsymbol{m}_d$：指示根据表面漫反射率而反射的入射光量。

7．$\boldsymbol{L}\cdot\boldsymbol{n}$：朗伯余弦定律。

8．α_h：中间向量$\boldsymbol{h}$与光向量$\boldsymbol{L}$之间的夹角。

9．$\boldsymbol{R}_F(\alpha_h)$：根据菲涅耳效应，关于中间向量$\boldsymbol{h}$所反射到观察者眼中的光量。

10．m：控制表面的粗糙度。

11．$(\boldsymbol{n}\cdot\boldsymbol{h})^m$：指定法线$\boldsymbol{h}$与宏观表面法线$\boldsymbol{n}$之间夹角为$\boldsymbol{\theta}_h$的所有微平面片段的分布情况（所占比例）。

12．$\frac{m+8}{8}$：在镜面反射过程中，为模拟能量守恒所采用的归一化因子。

图8.21展示了这3种分量协同工作的效果。

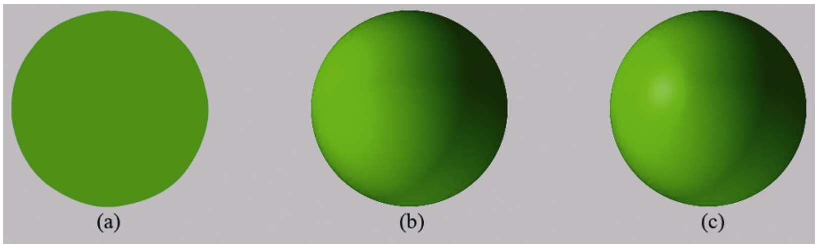


图8.21　图a中，仅采用环境光的球体着色效果，可以看出环境光以均匀的亮度照射着球体。图b中，环境光与漫反射光组合的光照效果。现在就可以观察到根据朗伯余弦定律所呈现出的由明到暗的平滑渐变。图c中，环境光、漫反射光以及镜面光所组合而成的最终光照效果。显而易见，镜面光照表现出了镜面高光。

注意　Note

式（8.4）是一种通用而又流行的光照方程，即便如此，它也仅是多种光照模型的其中之一而已。另有一些其他的光照模型，也值得读者前去探索一番。

8.9　材质的实现

我们所编写的材质结构体定义在d3dUtil.h文件中，部分代码如下：

```
// 在我们的演示程序中表示材质的简单结构体
struct Material
{
  // 便于查找材质的唯一对应名称
  std::string Name;

  // 本材质的常量缓冲区索引
  int MatCBIndex = -1;

  // 漫反射纹理在SRV堆中的索引。在第9章纹理贴图时会用到
  int DiffuseSrvHeapIndex = -1;

  // 已更新标志（dirty flag，也作脏标志）表示本材质已有变动，而我们也就需要更新常量缓冲区了。
  // 由于每个帧资源FrameResource都有一个材质常量缓冲区，所以必须对每个FrameResource都进
  // 行更新。因此，当修改某个材质时，应当设置NumFramesDirty = gNumFrameResources，以使每
  // 个帧资源都能得到更新
  int NumFramesDirty = gNumFrameResources;

  // 用于着色的材质常量缓冲区数据
  DirectX::XMFLOAT4 DiffuseAlbedo = { 1.0f, 1.0f, 1.0f, 1.0f };
  DirectX::XMFLOAT3 FresnelR0 = { 0.01f, 0.01f, 0.01f };
  float Roughness = 0.25f;
  DirectX::XMFLOAT4X4 MatTransform = MathHelper::Identity4x4();
};
```

为了模拟真实世界中的材质，需要设置DiffuseAlbedo（漫反射反照率）与FresnelR0（材质属性$\boldsymbol{R}_F(0°)$）这对与真实度相关的数值组合，再辅以一些关乎艺术性的细节调整。例如，金属导体吸收了进入金属内部的折射光[Möller08]，这就意味着，金属材质将不会发生漫反射（即DiffuseAlbedo的值为0）。然而，我们是不会100%按物理学上的

光照理论来建模的，而是稍作调整。一种富有艺术性的更佳策略是为`DiffuseAlbedo`设定一个非0的较小值。这一切取舍的关键点在于我们既应当尝试使用物理上接近现实的材质数值，也应该为艺术性留下一些空间，使之对数值适当调整，以展现出更佳的视觉效果。

在我们所用的材质结构体中，将粗糙度指定在归一化的浮点值范围[0, 1]内。粗糙度为0表示理想的光滑表面，粗糙度为1则表示实际能达到的最粗糙的表面。归一化范围使得不同材质之间粗糙度的比较更加方便。例如，粗糙度为0.6的材质比粗糙度为0.3的材质要加倍粗糙。在着色器代码中，我们将利用粗糙度来推导式（8.4）中所用的指数m。根据我们对粗糙度的定义可以发现，表面的光泽度（shininess，亦有译作反光度等）是与粗糙度相反的属性，$\text{shininess} = 1 - \text{roughness} \in [0,\ 1]$。

现在，摆在我们眼前的当务之急是该按照什么粒度（granularity）来指定材质的数据。材质的具体数值可能会随着表面而发生改变，即同一表面上不同点处的材质数据可能各不相同。例如，考虑图8.22所示的一辆汽车模型，其中的车身、车窗、车灯以及轮胎所反射和吸收的光量都是不同的，由此材质数值将随着汽车表面的位置而产生变化。

实现这种变化的解决方案之一是以每个顶点为基准来指定材质的具体数值。在三角形的光栅化处理期间，会对这些顶点中的材质属性进行插值计算，以求出三角形网格表面上每一点的材质数值。可是，就如我们在第7章的“Land and Waves”（陆地与波浪）演示程序中所见到的，逐顶点的颜色依然很“粗糙”，以致不能逼真地模拟出较为丰富的细节。除此之外，为了绘制每个顶点的颜色还要向顶点结构体中添加额外的数

据，同时，还需要采用不同的工具来绘制这些顶点颜色。事实上，更普遍的解决方法是采用纹理贴图。不过，这是第9章的主题。对于本章来说，我们在绘制调用时允许对材质进行频繁地更改。因此，我们为每种材质定义了唯一的属性，并将它们列于一个表中：

```
std::unordered_map<std::string, std::unique_ptr<Material>> mMaterials;

void LitWavesApp::BuildMaterials()
{
  auto grass = std::make_unique<Material>();
  grass->Name = "grass";
  grass->MatCBIndex = 0;
  grass->DiffuseAlbedo = XMFLOAT4(0.2f, 0.6f, 0.2f, 1.0f);
  grass->FresnelR0 = XMFLOAT3(0.01f, 0.01f, 0.01f);
  grass->Roughness = 0.125f;

  // 当前这种水的材质定义得并不是很好，但是由于我们还未学会所需的全部渲染工具（如
透明度、环境反
  // 射等），因此暂时先用这些数据解当务之急吧
  auto water = std::make_unique<Material>();
  water->Name = "water";
  water->MatCBIndex = 1;
  water->DiffuseAlbedo = XMFLOAT4(0.0f, 0.2f, 0.6f, 1.0f);
  water->FresnelR0 = XMFLOAT3(0.1f, 0.1f, 0.1f);
  water->Roughness = 0.0f;

  mMaterials["grass"] = std::move(grass);
  mMaterials["water"] = std::move(water);
}
```

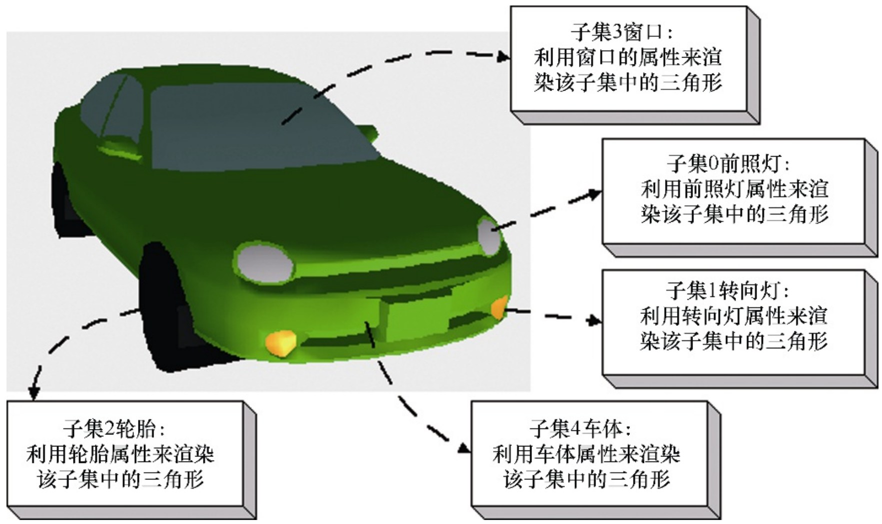


图8.22　一辆汽车的网格可以分成5种材质属性组

通过上面的表，可以将材质数据存放在系统内存之中。而为了令GPU能够在着色器中访问到这些材质数据，我们还需要将相关数据复制到常量缓冲区中。就像我们之前对物体常量缓冲区（per-object constant buffer）所做的一样，将存有每个材质常量的常量缓冲区添加到每个帧内资源FrameResource之中：

```
struct MaterialConstants
{
  DirectX::XMFLOAT4 DiffuseAlbedo = { 1.0f, 1.0f, 1.0f, 1.0f };
  DirectX::XMFLOAT3 FresnelR0 = { 0.01f, 0.01f, 0.01f };
  float Roughness = 0.25f;

  // 在纹理贴图章节中会用到
  DirectX::XMFLOAT4X4 MatTransform = MathHelper::Identity4x4();
};

struct FrameResource
{
public:
  ...

  std::unique_ptr<UploadBuffer<MaterialConstants>> MaterialCB =
    nullptr;
  ...
```

```
};
```

注意到结构体MaterialConstants中含有Material结构体内的部分数据，即着色器在渲染时所需的相关数据。

在更新函数中，当材质数据有了变化（即存在所谓的“脏数据”）时，便会将其复制到常量缓冲区的对应子区域内，因此GPU材质常量缓冲区中的数据总是与系统内存中的最新材质数据保持一致：

```
void LitWavesApp::UpdateMaterialCBs(const GameTimer& gt)
{
  auto currMaterialCB = mCurrFrameResource->MaterialCB.get();
  for(auto& e : mMaterials)
  {
    // 如果材质常量数据有了变化就更新常量缓冲区数据。一旦常量缓冲区数据发生改变，
就需对每一个帧
    // 资源FrameResource进行更新
    Material* mat = e.second.get();
    if(mat->NumFramesDirty > 0)
    {
      XMMATRIX matTransform = XMLoadFloat4x4(&mat->MatTransform);

      MaterialConstants matConstants;
      matConstants.DiffuseAlbedo = mat->DiffuseAlbedo;
      matConstants.FresnelR0 = mat->FresnelR0;
      matConstants.Roughness = mat->Roughness;

      currMaterialCB->CopyData(mat->MatCBIndex, matConstants);

      // 也需要对下一个FrameResource进行更新
      mat->NumFramesDirty--;
    }
  }
}
```

到现在为止，每一个渲染项都已含有一个指向Material结构体的指针了。注意，多个渲染项可以引用相同的Material对象，如多个渲染项能够使用相同的“板砖”材质。而每个Material对象都存有一个索

引，用于在材质常量缓冲区中指向它自己的常量数据。至此，我们就能在绘制渲染项时，找到对应常量数据的虚拟地址，并将它与所需材质常量数据的根描述符相绑定（其实也可以通过偏移到堆中的CBV描述符的方式来设置一个描述符表。不过，在此演示程序中，我们定义的根签名采用的是材质常量缓冲区的描述符，并非描述符表，所以这个方法在此例程中行不通）。下列代码演示了如何用不同的材质来绘制渲染项：

```
void LitWavesApp::DrawRenderItems(
      ID3D12GraphicsCommandList* cmdList,
      const std::vector<RenderItem*>& ritems)
  {
      UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize
       (sizeof(ObjectConstants));
      UINT matCBByteSize = d3dUtil::CalcConstantBufferByteSize
       (sizeof(MaterialConstants));

      auto objectCB = mCurrFrameResource->ObjectCB->Resource();
      auto matCB = mCurrFrameResource->MaterialCB->Resource();

      // 针对每个渲染项……
      for(size_t i = 0; i < ritems.size(); ++i)
      {
        auto ri = ritems[i];

        cmdList->IASetVertexBuffers(0, 1, &ri->Geo->VertexBufferView());
        cmdList->IASetIndexBuffer(&ri->Geo->IndexBufferView());
        cmdList->IASetPrimitiveTopology(ri->PrimitiveType);

        D3D12_GPU_VIRTUAL_ADDRESS objCBAddress =
          objectCB->GetGPUVirtualAddress() + ri->ObjCBIndex*objCBByteSize;
        D3D12_GPU_VIRTUAL_ADDRESS matCBAddress =
          matCB->GetGPUVirtualAddress() + ri->Mat->MatCBIndex*matCBByteSiz
e;

        cmdList->SetGraphicsRootConstantBufferView(0, objCBAddress);
        cmdList->SetGraphicsRootConstantBufferView(1, matCBAddress);

        cmdList->DrawIndexedInstanced(ri->IndexCount, 1,
           ri->StartIndexLocation, ri->BaseVertexLocation, 0);
      }
}
```

这里要着重说明的是，我们需要获取三角形网格表面上每一点处的法向量，以此来确定光线射向网格表面点处的角度（用于朗伯余弦定律）。而为了获取三角形网格表面上每个点处的近似法向量，我们就要在顶点这一层级来指定法线。在三角形的光栅化过程中，便会利用这些顶点法线进行插值计算。

到目前为止，我们已讨论过光的组成，但是还未讲到光源的具体类型。下面我们将描述如何来实现平行光源、点光源以及聚光灯光源。

8.10　平行光源

平行光源（parallel light）也称方向光源（directional light，也译作定向光源），是一种距离目标物体极远的光源。因此，我们就可以将这种光源发出的所有入射光看作是彼此平行的光线（见图8.23）。再者，由于光源距物体十分遥远，我们就能忽略距离所带来的影响，而仅指定照射到场景中光线的光强（light intensity）。

我们用向量来定义平行光源，借此即可指定光线传播的方向。因为这些光线是相互平行的，所以采用相同的方向向量代之。而光向量与光线传播的方向正相反。可以准确模拟出方向光的常见光源实例是太阳（见图8.24）。

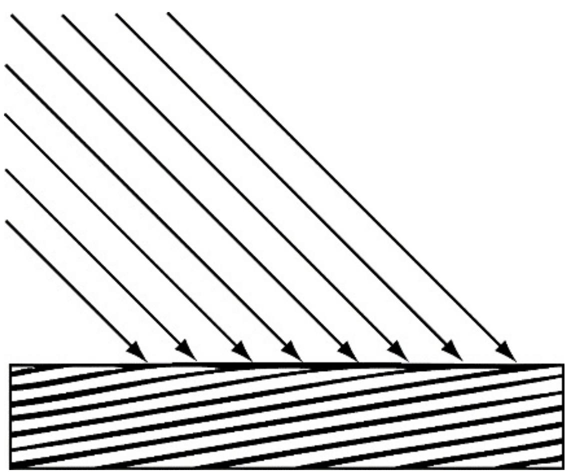


图8.23　照射到物体表面的平行光线

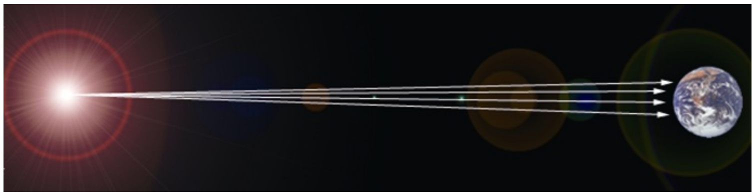


图8.24　这张图并没有按照实际比例绘制，但是如果选定的是地球表面上的一小块范围，那么照射到这块区域内的光线则近似于平行光

8.11　点光源

一个与点光源（point light）比较贴切的现实实例是灯泡，它能以球面向各个方向发出光线（见图8.25）。特别地，对于任意点$\boldsymbol{P}$，由位置$\boldsymbol{Q}$处点光源发出的光线，总有一束会传播至此点。像之前一样，我们定义光向量与光传播的方向相反，即光向量的方向是由点$\boldsymbol{P}$指向点光源$\boldsymbol{Q}$：

$$\boldsymbol{L}=\frac{\boldsymbol{Q}-\boldsymbol{P}}{\|\boldsymbol{Q}-\boldsymbol{P}\|}$$

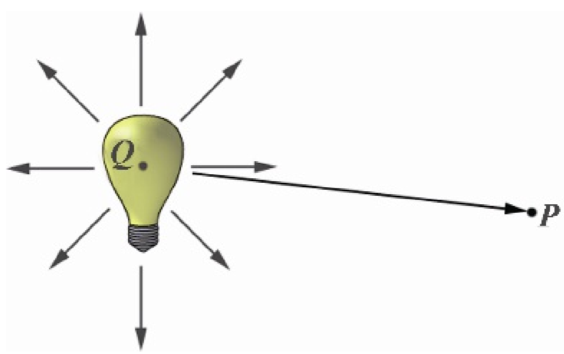


图8.25　点光源以球面向各个方向发出光线。特别是对于任意点$\boldsymbol{P}$，必存在一始于点光源$\boldsymbol{Q}$的光线照射到此点

其实点光与平行光之间唯一的区别就是光向量的计算方法——对于点光来说，光向量随着目标点的不同而改变；对于平行光而言，光向量则保持不变。

衰减

在物理学中，光强会根据平方反比定律（inverse squared law）而随着距离函数发生衰减。也就是说，距离光源d处的某点光强为：

$$I(d) = \frac{I_0}{d^2}$$

其中，I_0为距离光源$d = 1$处的光强。如果根据物理学来设置光量值（light value），并且辅之以HDR（high dynamic range，高动态范围）光照与色调映射（tonemapping，也可写作tone mapping，或译作色调贴图）技术，那效果定是极好的。然而，我们在这里却要使用一种更为简单的公式，并将它应用于演示代码之中。这便是我们所用的线性衰减（falloff）函数：

$$\text{att}(d) = \text{saturate}\left(\frac{\text{falloffEnd} - d}{\text{falloffEnd} - \text{falloffStart}}\right)$$

图8.26中所描绘的就是此线性衰减函数的图像。`saturate`函数会将它的参数限定在[0, 1]范围内：

$$\text{saturate}(x) = \begin{cases} x, 0 \leqslant x \leqslant 1 \\ 0, x < 0 \\ 1, x > 1 \end{cases}$$

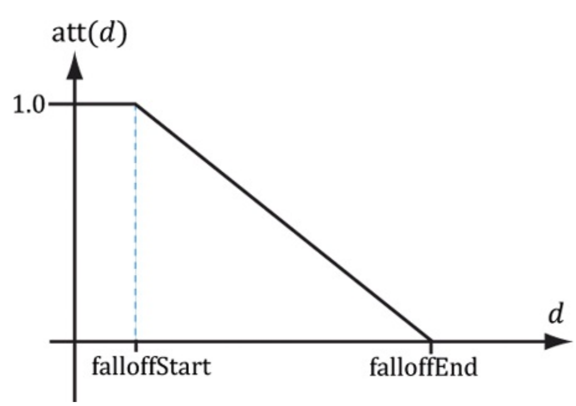


图8.26　在距离d未达falloffStart之前，衰减因子会一直令光量值保持最大强度（1.0）。当距离d到达falloffEnd时，衰减因子便会线性衰减至0.0

用此式计算点光光量与用式（8.4）计算点光光量的结果是相同的，但是千万别忘了，我们还须再将衰减因子$\mathrm{att}(d)$与光源的直射光量$\boldsymbol{B}_L$相乘。注意，衰减并不会影响环境光项，这是因为环境光项模拟的是向四处反弹后照射到目标物体的间接光。

在使用上述衰减函数的时候，若某点到光源的距离大于或等于falloffEnd则不会受到光照。这便为我们提供了一种极有用的光照优化手段：在着色器程序中，如果一个点超出了光照的有效范围，那么就可采用动态分支（dynamic branching）跳过此处的光照计算并提前返回。

8.12 聚光灯光源

一个与聚光灯光源（spotlight）相近的现实实例是手电筒。从本质上来说，聚光灯由位置$\boldsymbol{Q}$向方向$\boldsymbol{d}$照射出范围呈圆锥体的光（见图8.27）。

为了实现聚光灯光源，我们就要像在表示点光源时所做的一样，先指定光向量：

$$\boldsymbol{L}=\frac{\boldsymbol{Q}-\boldsymbol{P}}{\|\boldsymbol{Q}-\boldsymbol{P}\|}$$

其中，$\boldsymbol{P}$为被照点的位置，$\boldsymbol{Q}$是聚光灯的位置。观察图8.27可知，当且仅当位于$-\boldsymbol{L}$与$\boldsymbol{d}$之间的夹角ϕ小于圆锥体的半顶角ϕ_{max}时，$\boldsymbol{P}$才位于聚光灯的圆锥体范围之中（所以它才可以被光照射到）。而且，聚光灯圆锥体范围内所有光线的强度也不尽相同。位于圆锥体中心处的光线（即Qd这条向量上的光线）光强应该是最强的，而随着角ϕ由0增加至ϕ_{max}，光强会逐渐趋近于0。

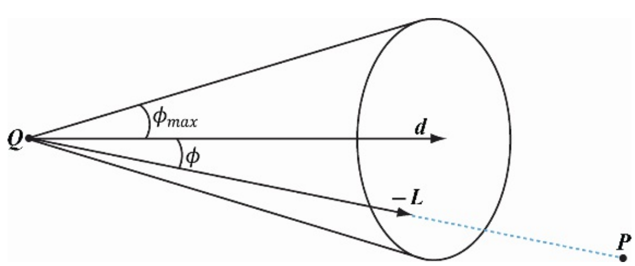


图8.27　一个聚光灯以位置$\boldsymbol{Q}$向方向$\boldsymbol{d}$发射出半顶角为$\phi_{\max}$的圆锥体范围的光

那么，怎样用与ϕ相关的函数来控制光强的衰减，又该如何来支配

聚光灯的圆锥体大小呢？答案是：我们可以使用与图8.19中曲线相同的函数，但是要以ϕ替换$\boldsymbol{\theta}_h$，再用s代替m：

$$k_{spot}(\phi) = \max(\cos\phi, 0)^s = \max(-\boldsymbol{L} \cdot \boldsymbol{d}, 0)^s$$

这正是我们所期待的公式：光强随着ϕ的增加而连续且平滑地衰减。另外，通过修改幂s，我们就能够间接地控制ϕ_{max}（使光强降为0的半顶角角度）。也就是说，我们可以通过设置不同的s值来缩小或扩大聚光灯光源的圆锥体照射范围。例如，如果设$s = 8$，则圆锥体的半顶角就逼近$45°$。

将光源的直射光量$\boldsymbol{B}_L$与衰减因子$\mathrm{att}(d)$相乘之后，还要根据被照射点位于聚光灯圆锥体的具体位置，用聚光灯因子k_{spot}按比例对光强进行缩放。除了上述操作之外，聚光灯的光照方程与式（8.4）相一致。

可以看出，使用聚光灯光源比使用点光光源的代价更高昂，因为我们需要额外计算聚光灯因子k_{spot}，并使之与聚光灯光强相乘。类似地，点光源又比方向光源的开销更高，因为点光需要针对距离d进行一系列计算（事实上，由于距离的计算涉及平方根运算，因此这个计算过程十分耗费资源），并且还需要求出衰减因子，再令它与点光光强相乘。总而言之，方向光是最廉价的光源，点光次之，最昂贵的光源则是聚光灯。

8.13 光照的具体实现

我们在本节中将对方向光、点光以及聚光灯的实现细节展开讨论。

8.13.1 Light结构体

在d3dUtil.h头文件中，我们定义了下列结构体来描述光源。此结构体可以表示方向光源、点光源与聚光灯光源。但是根据光源的具体类型，我们并不会用到其中的所有数据。比如说，在使用点光时就不会用到Direction（方向光）数据成员。

```
struct Light
{
  DirectX::XMFLOAT3 Strength = {0.5f, 0.5f, 0.5f};   // 光源的颜色
  float FalloffStart = 1.0f;                         // 仅供点光源/聚光灯光源使用
  DirectX::XMFLOAT3 Direction = {0.0f, -1.0f, 0.0f}; // 仅供方向光源/聚光灯光源使用
  float FalloffEnd = 10.0f;                          // 仅供点光源/聚光灯光源使用
  DirectX::XMFLOAT3 Position = { 0.0f, 0.0f, 0.0f }; // 仅供点光源/聚光灯光源使用
  float SpotPower = 64.0f;                           // 仅供聚光灯光源使用
};
```

文件LightingUtil.hlsl中则定义了与之对应的结构体：

```
struct Light
{
  float3 Strength;
  float FalloffStart; // 仅供点光源/聚光灯光源使用
  float3 Direction;   // 仅供方向光源/聚光灯光源使用
  float FalloffEnd;   // 仅供点光源/聚光灯光源使用
  float3 Position;    // 仅供点光源/聚光灯光源使用
```

```
  float SpotPower;    // 仅供聚光灯光源使用
};
```

结构体Light中数据成员的排列顺序并不是随意指定的（结构体MaterialConstants也是如此），这要遵从HLSL的结构体封装规则（structure packing rule）。详情可见附录B中的“常量缓冲区的封装规则”。这条HLSL规则的大意是以填充对齐的方式，将结构体中的元素打包为4D向量。另外，根据规则的限制，单个元素不能以一分为二的方式分到两个4D向量之中。这就意味着最好将上述结构体打包为3个4D向量，就像下面这样：

```
vector 1: (Strength.x, Strength.y, Strength.z, FalloffStart)
vector 2: (Direction.x, Direction.y, Direction.z, FalloffEnd)
vector 3: (Position.x, Position.y, Position.z, SpotPower)
```

从另一方面来说，如果将结构体写作：

```
struct Light
{
  DirectX::XMFLOAT3 Strength; // 光源的颜色
  DirectX::XMFLOAT3 Direction;// 仅供方向光源/聚光灯光源使用
  DirectX::XMFLOAT3 Position; // 仅供点光源/聚光灯光源使用
  float FalloffStart;         // 仅供点光源/聚光灯光源使用
  float FalloffEnd;           // 仅供点光源/聚光灯光源使用
  float SpotPower;            // 仅供聚光灯光源使用
};

struct Light
{
  float3 Strength;
  float3 Direction;   // 仅供方向光源/聚光灯光源使用
  float3 Position;    // 仅供点光源/聚光灯光源使用
  float FalloffStart; // 仅供点光源/聚光灯光源使用
  float FalloffEnd;   // 仅供点光源/聚光灯光源使用
  float SpotPower;    // 仅供聚光灯光源使用
};
```

那么，它将被打包为4个4D向量，如下：

```
vector 1: (Strength.x, Strength.y, Strength.z, empty)
vector 2: (Direction.x, Direction.y, Direction.z, empty)
vector 3: (Position.x, Position.y, Position.z, empty)
vector 4: (FalloffStart, FalloffEnd, SpotPower, empty)
```

可以看出，第二种方法占用了更多的空间，但这还是其次。此方法存在的更严重的问题是：在C++应用程序代码中，应有与HLSL部分相对应的结构体，但是C++结构体与HLSL结构体的封装规则并不相同。因此，若非小心地按HLSL封装法则来实现C++与HLSL的结构体，那么两者的结构体布局很有可能是不匹配的。如果这两种结构体不匹配，则通过`memcpy`函数从CPU上传至GPU常量缓冲区的数据将会导致渲染错误。

8.13.2 常用辅助函数

下面的3个函数定义于LightingUtil.hlsl文件中，由于这些代码可处理多种类型的光照，所以我们将其定义为辅助函数。

1. `CalcAttenuation`：实现了一种线性衰减因子的计算方法，可将其应用于点光源与聚光灯光源。

2. `SchlickFresnel`：代替菲涅耳方程的石里克近似。此函数基于光向量$\boldsymbol{L}$与表面法线$\boldsymbol{n}$之间的夹角，并根据菲涅耳效应近似地计算出以$\boldsymbol{n}$为法线的表面所反射光的百分比。

3．BlinnPhong：计算反射到观察者眼中的光量，该值为漫反射光量与镜面反射光量的总和。

```
float CalcAttenuation(float d, float falloffStart, float falloffEnd)
{
  // 线性衰减
  return saturate((falloffEnd-d) / (falloffEnd - falloffStart));
}

// 石里克提出的一种逼近菲涅耳反射率的近似方法
// (参见"Real-Time Rendering 3rd Ed."第233页)
// R0 = ( (n-1)/(n+1) )^2，式中的n为折射率
float3 SchlickFresnel(float3 R0, float3 normal, float3 lightVec)
{
  float cosIncidentAngle = saturate(dot(normal, lightVec));

  float f0 = 1.0f - cosIncidentAngle;
  float3 reflectPercent = R0 + (1.0f - R0)*(f0*f0*f0*f0*f0);

  return reflectPercent;
}

struct Material
{
  float4 DiffuseAlbedo;
  float3 FresnelR0;

  // 光泽度与粗糙度是一对性质相反的属性： Shininess = 1-roughness。
  float Shininess;
};

float3 BlinnPhong(float3 lightStrength, float3 lightVec,
         float3 normal, float3 toEye, Material mat)
{
  // m由光泽度推导而来，而光泽度则根据粗糙度求得
  const float m = mat.Shininess * 256.0f;
  float3 halfVec = normalize(toEye + lightVec);

  float roughnessFactor = (m + 8.0f)*pow(max(dot(halfVec, normal),
    0.0f), m) / 8.0f;
  float3 fresnelFactor = SchlickFresnel(mat.FresnelR0, halfVec, lightVec);

  // 尽管我们进行的是LDR（low dynamic range，低动态范围）渲染，但spec（镜面反
射）公式得到
  // 的结果仍会超出范围[0,1]，因此现将其按比例缩小一些
```

```
  specAlbedo = specAlbedo / (specAlbedo + 1.0f);

  return (mat.DiffuseAlbedo.rgb + specAlbedo) * lightStrength;
}
```

上述代码中所用的HLSL内部函数dot、pow与 max分别是向量点积函数、幂函数以及取最大值函数。大多数的HLSL内部函数描述可以从本书的附录B中找到。除此之外，那里还给出了一份HLSL的语法概览。有一点需要注意的是，当使用operator*令两个向量相乘时，即表示此乘法运算的计算方式是分量式乘法。

注意 Note

我们采用的镜面反照率计算公式允许其镜面值大于1，这表示非常耀眼的高光。然而，我们却希望渲染目标的颜色值在[0, 1]这个低动态范围内，因此一般来说将高于此范围的数值简单地钳制为1.0即可。但是，为了获得更加柔和的镜面高光就不宜“一刀切”式地钳制数值了，而是需要按比例缩小镜面反照率：

```
specAlbedo = specAlbedo / (specAlbedo + 1.0f);
```

高动态范围（HDR）光照使用的是光量值可超出范围[0, 1]的浮点渲染目标，在进行色调贴图这个步骤时，出于显示的目的会将高动态范围映射回[0, 1]区间，而在这个转换的过程中，保留细节信息是很重要的一项任务。HDR渲染与色调映射本身就是一门单独的学科——详见教材[Reinhard10]。而在另一篇文章 [Pettineo12] 中也给出了比较详尽的介

绍，还附有供读者用以实验的演示程序。

注意 Note

在计算机中，HLSL所用的全都是内联函数。因此，对于函数或传递参数而言并不会有过多的性能开销。

8.13.3 实现方向光源

给定观察位置$\boldsymbol{E}$、材质属性，与以$\boldsymbol{n}$为法线的表面上可见一点$\boldsymbol{p}$，则下列HLSL函数将输出自某方向光源发出，经上述表面以方向$\boldsymbol{v} = \text{normalize}(\boldsymbol{E} - \boldsymbol{p})$反射入观察者眼中的光量。在我们的示例中，此函数将由像素着色器所调用，以基于光照确定像素的颜色。

```
float3 ComputeDirectionalLight(Light L, Material mat, float3 normal,
float3 toEye)
{
  // 光向量与光线传播的方向刚好相反
  float3 lightVec = -L.Direction;

  // 通过朗伯余弦定律按比例降低光强
  float ndotl = max(dot(lightVec, normal), 0.0f);
  float3 lightStrength = L.Strength * ndotl;

  return BlinnPhong(lightStrength, lightVec, normal, toEye, mat);
}
```

8.13.4 实现点光源

给出观察点$\boldsymbol{E}$、以$\boldsymbol{n}$作为法线的表面上可视一点$\boldsymbol{p}$以及材质属性，则下面的HLSL函数将会输出从点光源放出，经上述表面在$\boldsymbol{v} = \text{normalize}(\boldsymbol{E} - \boldsymbol{p})$方向反射入观察者眼中的光量。在我们的示例中，该函数将被像素着色器所调用，并根据光照来确定像素的颜色。

```
float3 ComputePointLight(Light L, Material mat, float3 pos, float3 normal,

float3 toEye)
{
  // 自表面指向光源的向量
  float3 lightVec = L.Position - pos;

  // 由表面到光源的距离
  float d = length(lightVec);

  // 范围检测
  if(d > L.FalloffEnd)
    return 0.0f;

  // 对光向量进行规范化处理
  lightVec /= d;

  // 通过朗伯余弦定律按比例降低光强
  float ndotl = max(dot(lightVec, normal), 0.0f);
  float3 lightStrength = L.Strength * ndotl;

  // 根据距离计算光的衰减
  float att = CalcAttenuation(d, L.FalloffStart, L.FalloffEnd);
  lightStrength *= att;

  return BlinnPhong(lightStrength, lightVec, normal, toEye, mat);
}
```

8.13.5 实现聚光灯光源

指定观察点$\boldsymbol{E}$、以$\boldsymbol{n}$为法线的表面上可视一点$\boldsymbol{p}$以及材质属性，则下面的HLSL函数将会输出来自聚光灯光源，经过上述表面以方向$\boldsymbol{v}=\text{normalize}(\boldsymbol{E}-\boldsymbol{p})$反射入观察者眼中的光量。在我们的示例中，此函数将在像素着色器中被调用，以根据光照确定像素的颜色。

```
float3 ComputeSpotLight(Light L, Material mat, float3 pos, float3 normal,
float3 toEye)
{
  // 从表面指向光源的向量
  float3 lightVec = L.Position - pos;

  // 由表面到光源的距离
  float d = length(lightVec);

  // 范围检测
  if(d > L.FalloffEnd)
    return 0.0f;

  // 对光向量进行规范化处理
  lightVec /= d;

  // 通过朗伯余弦定律按比例缩小光的强度
  float ndotl = max(dot(lightVec, normal), 0.0f);
  float3 lightStrength = L.Strength * ndotl;

  // 根据距离计算光的衰减
  float att = CalcAttenuation(d, L.FalloffStart, L.FalloffEnd);
  lightStrength *= att;

  // 根据聚光灯照明模型对光强进行缩放处理
  float spotFactor = pow(max(dot(-lightVec, L.Direction), 0.0f), L.SpotPow
er);
  lightStrength *= spotFactor;

  return BlinnPhong(lightStrength, lightVec, normal, toEye, mat);
}
```

8.13.6　多种光照的叠加

光强是可以叠加的。因此，在支持多个光源的场景中，我们需要遍历每一个光源，并把它们在我们要计算光照的点或像素上的贡献值求和。示例框架最多可支持16个光源，凭此便可以用方向光、点光与聚光灯三种光源进行若干组合。当然，前提是光源的总数不能超过16个。此外，代码所采用的约定是方向光源必须位于光照数组的开始部分，点光源次之，聚光灯光源则排在末尾。下列代码用于计算某点处的光照方程：

```
#define MaxLights 16

// 绘制过程中所用的杂项常量数据
cbuffer cbPass : register(b2)
{
  ...
// 对于每个以MaxLights为光源数量最大值的对象来说，索引[0, NUM_DIR_LIGHTS)表示的是方向光源，
// 索引[NUM_DIR_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHTS)表示的是点光源
// 索引[NUM_DIR_LIGHTS+NUM_POINT_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHT+NUM_SPOT_
// LIGHTS) 则表示的是聚光灯光源

  Light gLights[MaxLights];
};

float4 ComputeLighting(Light gLights[MaxLights], Material mat,
            float3 pos, float3 normal, float3 toEye,
            float3 shadowFactor)
{
  float3 result = 0.0f;

  int i = 0;

#if (NUM_DIR_LIGHTS > 0)
  for(i = 0; i < NUM_DIR_LIGHTS; ++i)
  {
    result += shadowFactor[i] * ComputeDirectionalLight(gLights[i],
    mat, normal, toEye);
  }
#endif
```

```
#if (NUM_POINT_LIGHTS > 0)
  for(i = NUM_DIR_LIGHTS; i < NUM_DIR_LIGHTS+NUM_POINT_LIGHTS; ++i)
  {
    result += ComputePointLight(gLights[i], mat, pos, normal, toEye);
  }
#endif

#if (NUM_SPOT_LIGHTS > 0)
  for(i = NUM_DIR_LIGHTS + NUM_POINT_LIGHTS;
    i < NUM_DIR_LIGHTS + NUM_POINT_LIGHTS + NUM_SPOT_LIGHTS;
    ++i)
  {
    result += ComputeSpotLight(gLights[i], mat, pos, normal, toEye);
  }
#endif

  return float4(result, 0.0f);
}
```

可以观察到每种类型光源的数量实由多个#define来加以控制。这样一来，着色器将仅针对实际所需的光源数量来进行光照方程的计算。因此，如果一个应用程序只需3个光源，则我们仅对这3个光源展开计算。如果应用程序在不同阶段要支持不同数量的光源，那么只需生成以不同#define来定义的不同着色器即可。

注意 Note

参数shadowFactor在介绍阴影的章节中才会用到。我们暂时仅将它设置为向量(1, 1, 1)，这会使此阴影因子在方程中不会产生任何效果。

8.13.7 HLSL主文件

下面的代码含有本章演示程序中所用的顶点着色器与像素着色器，而其中所涉及的LightingUtil.hlsl文件中的HLSL代码，我们都已在此前进行了讨论。

```
//*********************************************************************
// Default.hlsl的作者为Frank Luna (C) 2015 版权所有
//
// 默认着色器，目前已支持光照
//*********************************************************************

// 光源数量的默认值
#ifndef NUM_DIR_LIGHTS
  #define NUM_DIR_LIGHTS 1
#endif

#ifndef NUM_POINT_LIGHTS
  #define NUM_POINT_LIGHTS 0
#endif

#ifndef NUM_SPOT_LIGHTS
  #define NUM_SPOT_LIGHTS 0
#endif

// 包含了光照所用的结构体与函数
#include "LightingUtil.hlsl"

// 每帧都有所变化的常量数据
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorld;
};

// 每种材质的不同常量数据
cbuffer cbMaterial : register(b1)
{
  float4 gDiffuseAlbedo;
  float3 gFresnelR0;
  float gRoughness;
  float4x4 gMatTransform;
};

// 绘制过程中所用的杂项常量数据
```

```
cbuffer cbPass : register(b2)
{
  float4x4 gView;
  float4x4 gInvView;
  float4x4 gProj;
  float4x4 gInvProj;
  float4x4 gViewProj;
  float4x4 gInvViewProj;
  float3 gEyePosW;
  float cbPerObjectPad1;
  float2 gRenderTargetSize;
  float2 gInvRenderTargetSize;
  float gNearZ;
  float gFarZ;
  float gTotalTime;
  float gDeltaTime;
  float4 gAmbientLight;

  // 对于每个以MaxLights为光源数量最大值的对象来说，索引[0, NUM_DIR_LIGHTS)表
示的是方向光
  // 源，索引[NUM_DIR_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHTS)表示的是点光源
  // 索引[NUM_DIR_LIGHTS+NUM_POINT_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHT+
NUM_SPOT_
  // LIGHTS)表示的是聚光灯光源

  Light gLights[MaxLights];
};

struct VertexIn
{
  float3 PosL   : POSITION;
  float3 NormalL : NORMAL;
};

struct VertexOut
{
  float4 PosH   : SV_POSITION;
  float3 PosW   : POSITION;
  float3 NormalW : NORMAL;
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout = (VertexOut)0.0f;

  // 将顶点变换到世界空间
  float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
```

```
  vout.PosW = posW.xyz;

  // 假设这里进行的是等比缩放，否则这里需要使用世界矩阵的逆转置矩阵
  vout.NormalW = mul(vin.NormalL, (float3x3)gWorld);

  // 将顶点变换到齐次裁剪空间
  vout.PosH = mul(posW, gViewProj);

  return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
  // 对法线插值可能导致其非规范化，因此需要再次对它进行规范化处理
  pin.NormalW = normalize(pin.NormalW);

  // 光线经表面上一点反射到观察点这一方向上的向量
  float3 toEyeW = normalize(gEyePosW - pin.PosW);

  // 间接光照
  float4 ambient = gAmbientLight*gDiffuseAlbedo;

  // 直接光照
  const float shininess = 1.0f - gRoughness;
  Material mat = { gDiffuseAlbedo, gFresnelR0, shininess };
  float3 shadowFactor = 1.0f;
  float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
    pin.NormalW, toEyeW, shadowFactor);

  float4 litColor = ambient + directLight;

  // 从漫反射材质中获取alpha值的常见手段
  litColor.a = gDiffuseAlbedo.a;

  return litColor;
}
```

8.14　光照演示程序

本光照演示程序是在第7章“Land and Waves”例程的基础之上构建而成的。其中利用了一个方向光来表示太阳。用户可以用左、右、上、下4个方向键来控制太阳的方位。由于我们已经讨论过如何实现材质与光源，所以下面的各小节中仅处理尚未实现的部分细节。图8.28是一张光照演示程序的效果。

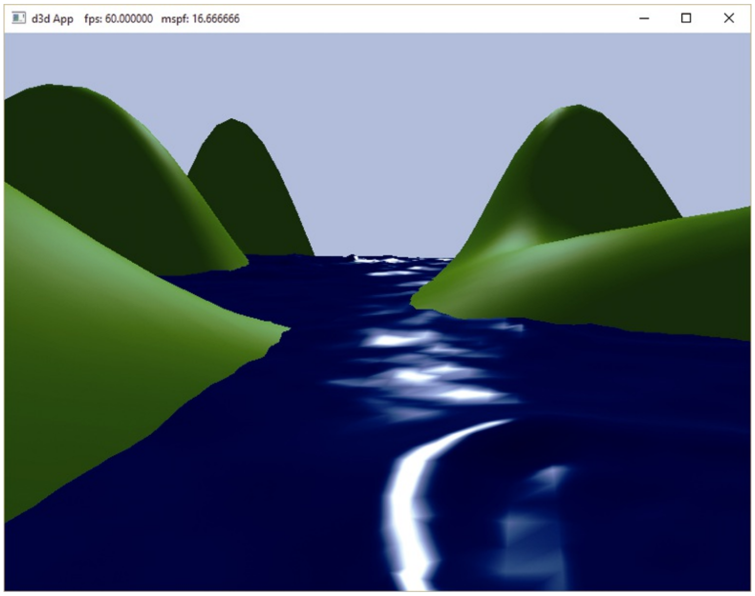


图8.28　光照演示程序的屏幕效果

8.14.1　顶点格式

光照的计算依赖于表面法线。我们在顶点层级定义了法线，借此对位于三角形中的每个像素都进行插值计算，由此展开逐像素光照。另外，我们也不再指定顶点的颜色，而是以每个像素应用光照方程后所生

成的像素颜色加以代替。为了支持顶点法线，我们将之前的顶点结构修改如下：

```
// C++顶点结构体
struct Vertex
{
  DirectX::XMFLOAT3 Pos;
  DirectX::XMFLOAT3 Normal;
};

// 对应的HLSL顶点结构体
struct VertexIn
{
  float3 PosL   : POSITION;
  float3 NormalL : NORMAL;
};
```

当修改了顶点格式后，我们就要随之更新输入布局描述来对比进行说明：

```
mInputLayout =
{
  { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
   D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
  { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
   D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }
};
```

8.14.2 计算法线

GeometryGenerator类中用于生成各种几何形状的函数，已能够通过顶点法线去创建对应的图形数据，可谓万事俱备。然而，由于我们为了令地形（terrain）表面更加真实而修改了此演示程序中的栅格高度，所以还需为地形生成法向量。

因为地形曲面由函数$y=f(x,z)$给出，所以我们可以通过微积分知识来直接计算法向量，而不必再用8.2.1节中所述的求法线平均值方法。为此，针对曲面上的每一个点，我们都通过偏导数在+x与+z方向上建立两个切向量（tangent vector）：

$$\boldsymbol{T}_X=\left(1,\frac{\partial f}{\partial x},0\right)$$
$$\boldsymbol{T}_Z=\left(1,\frac{\partial f}{\partial z},0\right)$$

这两个向量都位于曲面点的切平面上，求这两个向量的叉积即可得到对应点处的法向量：

$$\begin{aligned}\boldsymbol{n}=\boldsymbol{T_Z}\times\boldsymbol{T}_X&=\begin{vmatrix}\boldsymbol{i}&\boldsymbol{j}&\boldsymbol{k}\\0&\frac{\partial f}{\partial z}&1\\1&\frac{\partial f}{\partial x}&0\end{vmatrix}\\&=\left(\begin{vmatrix}\frac{\partial f}{\partial z}&1\\\frac{\partial f}{\partial x}&0\end{vmatrix},-\begin{vmatrix}0&1\\1&0\end{vmatrix},\begin{vmatrix}0&\frac{\partial f}{\partial z}\\1&\frac{\partial f}{\partial x}\end{vmatrix}\right)\\&=\left(-\frac{\partial f}{\partial x},1,-\frac{\partial f}{\partial z}\right)\end{aligned}$$

用来生成陆地网格的函数为：

$$f(x,z)=0.3z\cdot\sin(0.1x)+0.3x\cdot\cos(0.1z)$$

则偏导数为：

$$\frac{\partial f}{\partial x}=0.03z\cdot\cos(0.1x)+0.3\cos(0.1z)$$
$$\frac{\partial f}{\partial z}=0.3\sin(0.1x)-0.03x\cdot\sin(0.1z)$$

而位于曲面上一点$(x,f(x,z),z)$处的曲面法线为：

$$\boldsymbol{n}(x, z) = \left(-\frac{\partial f}{\partial x}, 1, -\frac{\partial f}{\partial z}\right) = \begin{bmatrix} -0.03z \cdot \cos(0.1x) - 0.3\cos(0.1z) \\ 1 \\ -0.3\sin(0.1x) + 0.03x \cdot \sin(0.1z) \end{bmatrix}^{\mathrm{T}}$$

我们可以注意到曲面法线并不具有单位长度，所以在光照计算之前还需对它进行规范化处理。

我们要在每个顶点处都进行上述的法线计算，以获取对应的顶点法线：

```
XMFLOAT3 LitWavesApp::GetHillsNormal(float x, float z)const
{
  // n = (-df/dx, 1, -df/dz)
  XMFLOAT3 n(
    -0.03f*z*cosf(0.1f*x) - 0.3f*cosf(0.1f*z),
    1.0f,
    -0.3f*sinf(0.1f*x) + 0.03f*x*sinf(0.1f*z));

  XMVECTOR unitNormal = XMVector3Normalize(XMLoadFloat3(&n));
  XMStoreFloat3(&n, unitNormal);

  return n;
}
```

水面的法向量与陆地表面法向量的求法相似，只是我们还没有找到用于计算水面的相应公式。但是，每个顶点处的切向量都可用有限差分格式（finite difference scheme）来近似计算（参见[Lengyel02]或其他与数值分析有关的书籍）。

注意 Note

如果读者对微积分方面的知识比较生疏，也不必担心，因为这并不

是本书的重头戏。它现在显得比较重要的原因是我们需要用数学描述的曲面来生成几何体，以此来绘制一些有趣的物体。在后面的章节中，我们会改为从文件中加载由3D建模程序导出的3D网格。

8.14.3　更新光照的方向

正如8.13.7节中所述，`Light`数组被置于渲染过程常量缓冲区（per-pass constant buffer）中。演示程序使用了一个方向光源来表示太阳，并允许用户通过左、右、上、下4个方向键来控制光源的方位。这就是说，我们需要在每一帧中都重新计算阳光照射的方向，并将结果设置到渲染过程常量缓冲区内。

我们用球坐标(ρ,θ,ϕ)来追踪太阳的位置，但是，由于假设太阳距此是无限远的，所以径向距离ρ的取值是无关紧要的。这里设$\rho=1$，以此令太阳位于单位球面这一轨道之上，即用$(1,\theta,\phi)$来表示太阳的朝向。而光的方向与太阳的朝向正相反。下列是用于更新太阳方位的相关代码。

```
float mSunTheta = 1.25f*XM_PI;
float mSunPhi = XM_PIDIV4;

void LitWavesApp::OnKeyboardInput(const GameTimer& gt)
{
  const float dt = gt.DeltaTime();

  if(GetAsyncKeyState(VK_LEFT) & 0x8000)
    mSunTheta -= 1.0f*dt;

  if(GetAsyncKeyState(VK_RIGHT) & 0x8000)
    mSunTheta += 1.0f*dt;
```

```
  if(GetAsyncKeyState(VK_UP) & 0x8000)
    mSunPhi -= 1.0f*dt;

  if(GetAsyncKeyState(VK_DOWN) & 0x8000)
    mSunPhi += 1.0f*dt;

  mSunPhi = MathHelper::Clamp(mSunPhi, 0.1f, XM_PIDIV2);
}

void LitWavesApp::UpdateMainPassCB(const GameTimer& gt)
{
    ...
  XMVECTOR lightDir = -MathHelper::SphericalToCartesian(1.0f,
    mSunTheta, mSunPhi);

  XMStoreFloat3(&mMainPassCB.Lights[0].Direction, lightDir);
  mMainPassCB.Lights[0].Strength = { 1.0f, 1.0f, 0.9f };

  auto currPassCB = mCurrFrameResource->PassCB.get();
  currPassCB->CopyData(0, mMainPassCB);
}
```

注意 Note

将`Light`数组放在渲染过程常量缓冲区中，就意味着在渲染过程中所用的光源不能超过16个（此值是程序能支持的最多光源数量）。这对小的演示程序来说没有什么影响。但是对于大型的游戏场景而言，这是远远不够的，我们应该可以想象得出：在游戏这一级别上的应用，会用到数以百计的光源遍布场景。而对于这种情况的一种解决方案是将`Light`数组置于物体常量缓冲区（per-object constant buffer）中。接下来，针对每个物体O，我们可以在场景中搜寻会作用于它的光源，并将这些光源与此物体的常量缓冲区相绑定。如此一来，这些光源便会以它的光照范围（点光源是球体而聚光灯光源是圆锥体）来照射物体O。除

此之外，另一种常见的策略是使用延迟渲染（deferred rendering）或Forward+渲染等技术。

8.14.4 更新根签名

为了实现光照，我们为着色器程序引入了一个新的材质常量缓冲区。为了支持这个新添加的常量缓冲区，就需要改写之前的根签名。正如物体常量缓冲区（per-object constant buffer）一样，对于材质常量缓冲区我们也使用了相应的根描述符，这样便可以绕过描述符堆而直接绑定常量缓冲区。

8.15　小结

1．运用光照后，我们就不再指定每个顶点的颜色，取而代之的是要定义场景光源与每个顶点的材质。可将材质看作是确定光与物体表面如何互相交互的一种属性。通过对三角形表面每个顶点处的材质进行插值，即可获取三角形网格中每一个表面点处的材质数据。光照方程则根据光与表面材质之间的交互来计算观察者所见到的表面颜色。除此之外，光照方程还涉及一些其他的参数，如表面法线与观察点。

2．曲面法线（surface normal）是一种正交于曲面上某点切平面的单位向量。利用曲面法线可确定曲面上某点的“朝向”。针对光照计算，我们需要求出三角形网格曲面上每一点处的曲面法线，以此来确定光线照射到网格表面点处的角度。为了获取曲面法线，我们仅指定了顶点处的表面法线（vertex normal，所以也称为顶点法线）。接下来，为了获取三角形网格曲面上每一点处的曲面近似法线，需要在光栅化期间对三角形中的这些顶点法线进行插值。对于任意三角形网格来说，一般要通过一种叫作求法线平均值的计算方法来估算顶点法线。如果矩阵$\boldsymbol{A}$可用于变换点与向量（非法向量），那么$(\boldsymbol{A}^{-1})^{\mathrm{T}}$便可用于变换经非等比变换或剪切变换后的法线。

3．平行光源（方向光）模拟了一种距被照物体极远的光源。据此，我们可以将入射光线看作是彼此平行的光线。方向光的一个现实实例是太阳向地球发射的光。点光源会向四周各个方向发光，它的一个现实示例是电灯泡。聚光灯光源的发光范围是圆锥体，实际生活中的例子

是家用电器手电筒。

4．根据菲涅耳效应可知，当光到达两种不同折射率介质之间的界面时，一部分光会被反射，其余的光则会折射入介质。反射的光量依赖于介质（某些材质的反射率要偏高一些）以及法向量$\boldsymbol{n}$与光向量$\boldsymbol{L}$之间的夹角θ_i。考虑到这个过程的复杂性，以致完整的菲涅耳方程一般不会用于实时渲染，而是采用石里克近似（Schlick approximation）加以代替。

5．现实世界中的反射物体一般不是理想镜面。纵然一个物体的表面看起来十分平滑，但在微观水平上来看它还是有一定的粗糙度（roughness）。我们可以认为理想的镜面粗糙度为0，并且它的微观表面法线都与宏观表面法线指向相同的方向。随着粗糙度的增加，微观表面法线纷纷偏离宏观表面法线，并导致反射光逐渐扩展为镜面瓣（specular lobe）。

6．环境光模拟了在场景中进行多次散射与反弹，并且按各个方向均等射向物体的间接光，因此它提供的是均匀光照。漫反射光模拟的是进入介质内部的光，它将在表面下进行散射，其中的一部分会被吸收，而剩下的光则将散射回表面。由于对表面下的散射过程建模比较困难，我们便假设从介质返回来的光在表面上光的入射点处向所有方向均等地散射。镜面光则模拟了根据菲涅耳效应与表面粗糙度而从表面反射的光。

8.16 练习

1．修改本章的光照演示程序，使方向光源仅发出偏红色的光。此外，通过使用正弦函数令光的强度随时间函数而有规律地改变，继而使光按脉冲形式（明暗渐变）得以展现。利用带有特定色彩的闪烁光源可以很好地渲染出不同的游戏氛围，例如，闪烁的红光可用于表现气氛紧张的急救场面。

2．修改本章光照演示程序中的材质粗糙度。

3．修改前一章中的“Shapes”（多种不同的几何图形）演示程序，向其添加材质以及三点布光（three-point lighting，也译作三点打光、三点光照等）系统。三点布光系统常用于电影与摄影工作，它可以提供比单光源更佳的光照效果。该系统由三种光源构成，即称为主光（key light）的主要光源、通常在主光的侧面对准目标的辅助光（fill light，也称补光）以及轮廓光（back light，又称背光）。我们用三点布光的方式充当间接光照，可以比仅使用构成间接光照的环境光更好地诠释目标对象。在本示例中，请用3个方向光来构建三点布光系统，见图8.29。

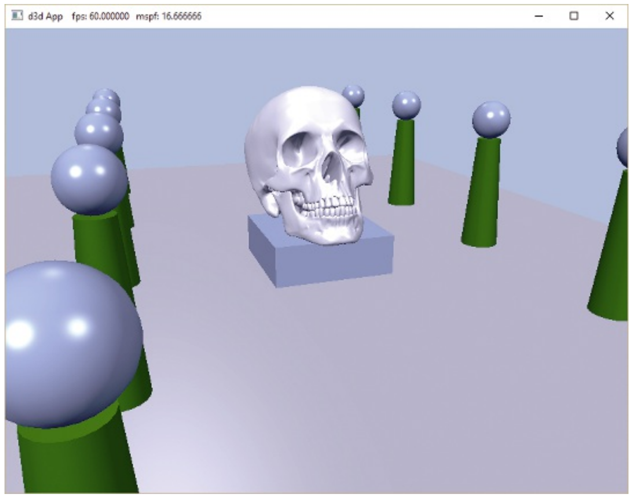


图8.29　练习3解答方案的效果

4．通过撤去三点布光，并在每个柱子上的球体中心添加一个点光源来修改练习3的解决方案。

5．移除三点布光系统，通过向每个柱子上球体的中心添加一个聚光灯光源，并令它们向偏下方打光以修改练习3的解决方案。

6．卡通风格光照的一个特点是颜色之间的过渡比较突兀（与大家通常喜闻乐见的平滑渐变刚好相反），从而塑造夸张的效果，如图8.30所示。这种光照一般可以通过计算k_d与k_s来得以实现，但是在像素着色器中使用这种光照之前，先要运用下列离散函数对这两个参数进行变换：

$$k'_d = f(k_d) = \begin{cases} 0.4 \text{当} -\infty < k_d \leqslant 0.0\text{时} \\ 0.6 \text{当} \ 0.0 < k_d \leqslant 0.5\text{时} \\ 1.0 \text{当} \ 0.5 < k_d \leqslant 1.0\text{时} \end{cases}$$

$$k'_s = g(k_s) = \begin{cases} 0.0 \text{当} \ 0.0 < k_s \leqslant 0.1\text{时} \\ 0.5 \text{当} \ 0.1 < k_s \leqslant 0.8\text{时} \\ 0.8 \text{当} \ 0.8 < k_s \leqslant 1.0\text{时} \end{cases}$$

请用这种卡通着色的方式来修改本章的光照演示程序。（注意，上述函数f与g只是便于上手的示例函数，我们可以根据预期的结果而自行调整。其中，$k_{\mathrm{d}}=\max(\boldsymbol{L}\cdot\boldsymbol{n},0)$，$k_{\mathrm{s}}=\begin{cases}\max(\boldsymbol{n}\cdot\boldsymbol{h},0)^m, & \boldsymbol{L}\cdot\boldsymbol{n}>0\\ 0, & \boldsymbol{L}\cdot\boldsymbol{n}\leqslant 0\end{cases}$.）再将$k_d$与$k_s$代回式8.4即可求出光照的颜色值。其实关键点就在于函数f与g，它们将原本的渐变过程减少为上述的若干种组合。

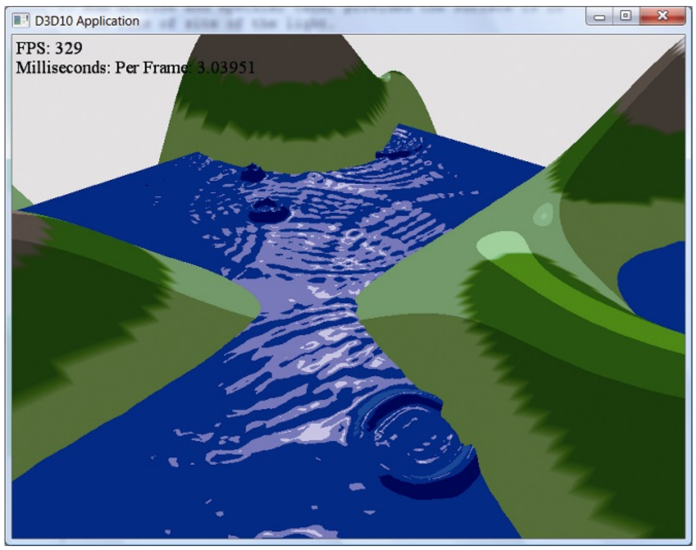


图8.30　卡通光照效果的效果截图

---

[1]　normalize即规范化之意。在代码中与之对应的是HLSL中的内置规范化函数`normalize`。

[2]　即clamp，该词常用于电子学中，此处引申为将某值限制特定范围之内，也有译作截断。

[3]　要计算出真实的光照效果，要以现实世界中的物理为基础展开讨论。光具有波粒二象性，因而可由此展开。根据文中的所用词汇可知

（辐射通量），本书是以辐（射）度学的角度来阐述光照的。就像其他表示光量的因素一样，漫反射反照率最终反映的其实是种颜色，并在程序中以一个计算项的身份参与光照的计算。而计算光照也就是计算最终显示出来的颜色。

第9章　纹理贴图

我们的演示程序开始变得越发有趣了，但是，这现实世界中的物体往往却要比物体常量缓冲区中可捕获的材质有着更加丰富的细节。纹理贴图（texture mapping，也译作纹理映射）就是这样一种将图像数据映射到网格三角形上的技术，可以使我们为场景增添更多的细节，令它更具真实感。例如，我们可以构建一个立方体，再将板条箱的纹理映射到它的每个面上，使它看起来像个木质的货物箱（见图9.1）。

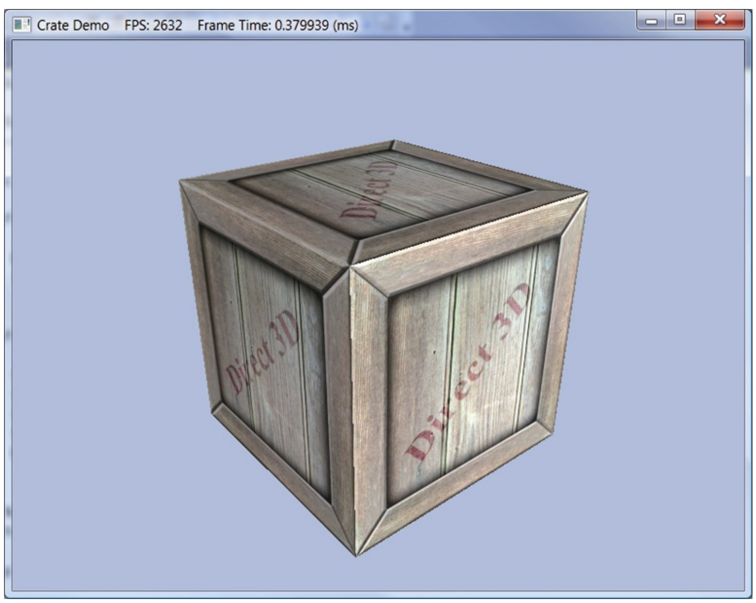


图9.1　以映射有板条箱纹理的立方体创建的货物箱例程

学习目标：

1．学习如何将局部纹理映射到网格三角形上。

2．探究如何创建和启用纹理。

3．学会如何通过纹理过滤来创建更为平滑的图像效果。

4．探索如何用寻址模式来进行多次贴图。

5．探究如何将多个纹理进行组合，从而创建出新的纹理与特效。

6．学习如何通过纹理动画来创建一些基本效果。

9.1　纹理与资源的回顾

回顾前文可知，其实我们自第4章起就已经开始使用纹理了。特别是深度缓冲区与后台缓冲区，它们都是通过ID3D12Resource接口表示，并以D3D12_RESOURCE_DESC::Dimension成员中的D3D12_RESOURCE_DIMENSION_TEXTURE2D类型来描述的2D纹理对象。为了便于参考，在本节中，我们先来重温在第4章里已经讨论过的与纹理相关的知识。

2D纹理是一种由特定数据元素所构成的矩阵[1]，它的用处之一即是存储2D图像数据，纹理中的每个元素都存储着对应像素的颜色。但是，这并不是它唯一的用途，比如说，在一种称为法线贴图的高级技术中，每个纹理元素都存储的是一个3D向量而非颜色数据。因而，尽管一提及纹理给人的第一印象是用以存储图像数据的资源，但实际上它具有更广泛的用处。1D纹理（D3D12_RESOURCE_DIMENSION_TEXTURE1D）与3D纹理（D3D12_RESOURCE_DIMENSION_TEXTURE3D）就像由数据元素构成的1D、3D数组。而1D、2D以及3D纹理实则都用泛型接口ID3D12Resource来表示。

纹理不同于缓冲区资源，因为缓冲区资源仅存储数据数组，而纹理却可以具有多个mipmap层级（后文有介绍），GPU会基于这个层级进行相应的特殊操作，例如运用过滤器以及多重采样。支持这些特殊操作纹理的资源都被限定为一些特定的数据格式。而缓冲区资源就没有这项

限制，它们可以存储任意类型的数据。纹理所支持的数据格式由枚举类型DXGI_FORMAT来表示。这是其中的一些格式示例：

1．DXGI_FORMAT_R32G32B32_FLOAT：每个元素由3个32位浮点数分量构成。

2．DXGI_FORMAT_R16G16B16A16_UNORM：每个元素由4个16位分量组成，每个分量都将被映射到范围[0, 1]之间。

3．DXGI_FORMAT_R32G32_UINT：每个元素由2个32位无符号整数分量构成。

4．DXGI_FORMAT_R8G8B8A8_UNORM：每个元素由4个8位无符号分量构成，每个分量都将被映射到范围[0, 1]之间。

5．DXGI_FORMAT_R8G8B8A8_SNORM：每个元素由4个8位有符号分量构成，每个分量都将被映射到范围[-1, 1]之间。

6．DXGI_FORMAT_R8G8B8A8_SINT：每个元素由4个8位有符号整数分量构成，每个分量都将被映射到范围[-128, 127]之间。

7．DXGI_FORMAT_R8G8B8A8_UINT：每个元素由4个8位无符号整数分量构成，每个分量都将被映射到范围[0, 255]之间。

顾名思义，R、G、B、A这4个字母分别用于表示红色、绿色、蓝色以及alpha值。然而，正如前面所说，纹理不一定要存储颜色信息。例如，格式

```
DXGI_FORMAT_R32G32B32_FLOAT
```

具有3个浮点数分量，因此它能够存储一个3D向量的浮点坐标（并不是颜色向量）。还有一种无类型（typeless）格式，我们仅用它来预留一块内存，并要在稍后将纹理绑定到渲染流水线的时候指出如何重新解释其中的数据（有点像类型强制转换）。例如，下列无类型格式就为每个元素预留了4个8位分量，但是并没有指定其具体的数据类型（如整数、浮点数、无符号整数等）：

```
DXGI_FORMAT_R8G8B8A8_TYPELESS
```

注意 Note

DirectX 11 SDK文档中提到：“以某种具体类型创建的资源，其格式是不能更改的。这将使该资源在运行时的访问得以优化 [……]。”因此，我们应当只在不得已的情况下才使用无类型资源，否则就用具体的类型来创建资源。

一个纹理可以绑定到渲染流水线的不同阶段，一个常见的例子是既可将一纹理用作渲染目标（即Direct3D中的渲染到纹理技术），又能把它作为着色器资源（即在着色器中对该纹理进行采样）。一个纹理可以当作渲染目标，也可以充当着色器资源，但是不能同时“身兼数职”。将数据渲染到一个纹理后，再用它作为着色器资源，这种方法称为渲染到

纹理（render-to-texture），本书的后面会用此技术来实现一些有趣的特效。要使纹理扮演渲染目标与着色器资源这两种角色，我们就需要为此纹理资源创建两个描述符：一个存于渲染目标堆中（以D3D12_DESCRIPTOR_HEAP_TYPE_RTV描述），另一个位于着色器资源堆中（以D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV描述。注意，从着色器资源堆的枚举名称中能够看出，它也可以存储常量缓冲区视图描述符以及无序访问视图描述符）。接下来，我们便可以把该资源绑定为渲染目标，或者作为着色器的输入与根签名中的根参数相绑定（当然，这两种身份并不能同时共存）：

```
// 绑定为渲染目标
CD3DX12_CPU_DESCRIPTOR_HANDLE rtv = ...;
CD3DX12_CPU_DESCRIPTOR_HANDLE dsv = ...;
cmdList->OMSetRenderTargets(1, &rtv, true, &dsv);

// 以着色器输入的名义绑定到根参数
CD3DX12_GPU_DESCRIPTOR_HANDLE tex = ...;
cmdList->SetGraphicsRootDescriptorTable(rootParamIndex, tex);
```

资源描述符实际上做了两件事：它们通知Direct3D这些资源将被如何使用（即我们将资源绑定到流水线的哪个阶段）。如果以无类型格式来创建资源，那么我们一定要在为它创建视图时指定其具体类型。这样一来，若使用了无类型格式，我们就能在某个流水线阶段中将纹理的元素视为浮点值，而在另一流水线步骤中将该纹理的元素视为整数，这一过程其实就相当于对数据进行强制转换。

在本章中，我们只探讨将纹理绑定为着色器资源，据此，像素着色器就能对纹理进行采样并处理其中表示颜色的像素数据。

9.2　纹理坐标

Direct3D所采用的纹理坐标系，是由指向图像水平正方向的u轴与指向图像垂直正方向的v轴所组成的。取值范围为$0 \leqslant u, v \leqslant 1$的坐标$(u, v)$标定的是一种称为纹素（texel）的纹理元素。观察图9.2可以发现，v轴的正方向指向图像的“正下”方。此外，还可以看出纹理坐标采用的是归一化坐标区间[0, 1]，通过这种方式便可令Direct3D的工作摆脱具体纹理尺寸的干扰。例如，无论纹理实际的大小是256×256像素、512×1024像素还是2048×2048像素，纹理坐标(0.5, 0.5)总是表示纹理正中间的纹素；坐标(0.25, 0.75)总是表示在纹理中，位于水平方向总宽度1/4、垂直方向总高度3/4处的纹素。目前所讨论的纹理坐标都是在区间[0, 1]之内，在后面，我们会解释若超出了此范围会发生什么。

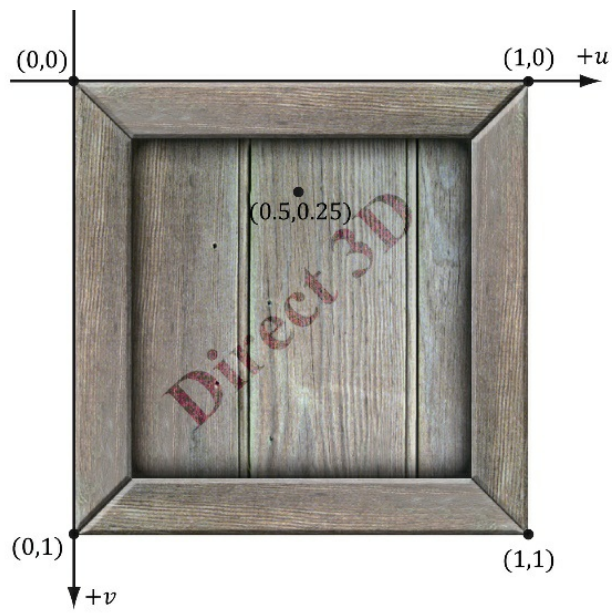


图9.2　图中所示的是纹理坐标系，有时也被称为纹理空间[2]

对于每个3D三角形来说，我们希望在将要映射于其上的纹理中定义出与之对应的三角形（见图9.3）。设$\boldsymbol{p}_0$、$\boldsymbol{p}_1$以及$\boldsymbol{p}_2$为3D三角形的3个顶点，它们分别对应于纹理坐标$\boldsymbol{q}_0$、$\boldsymbol{q}_1$与$\boldsymbol{q}_2$。针对3D三角形上任意一点(x,y,z)处的纹理坐标(u,v)，我们都可以通过与3D三角形坐标插值所用的相同参数s、t，对顶点纹理坐标进行线性插值来求得。这就是说，如果：

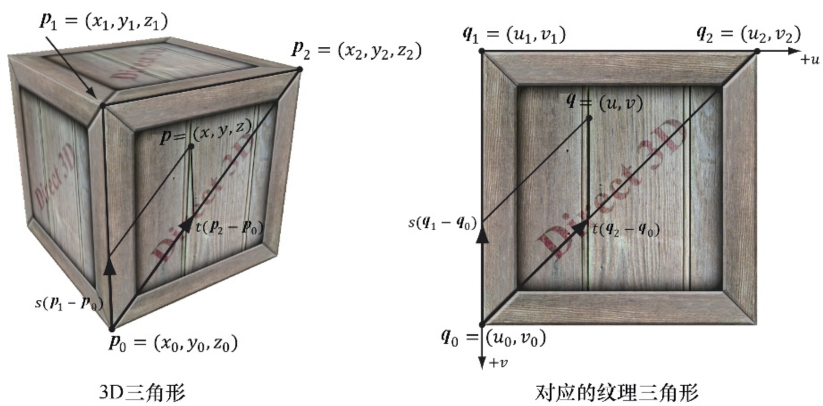


图9.3　左侧的三角形位于3D空间，我们将把右侧纹理上的2D三角形映射到左侧的3D三角形上

$$(x,y,z) = \boldsymbol{p} = \boldsymbol{p}_0 + s(\boldsymbol{p}_1 - \boldsymbol{p}_0) + t(\boldsymbol{p}_2 - \boldsymbol{p}_0)$$

当$s \geqslant 0, t \geqslant 0, s+t \leqslant 1$时，那么，

$$(u,v) = \boldsymbol{q} = \boldsymbol{q}_0 + s(\boldsymbol{q}_1 - \boldsymbol{q}_0) + t(\boldsymbol{q}_2 - \boldsymbol{q}_0)$$

依此方法便可求出三角形上每个点处的对应纹理坐标。

为了实现此计算过程，我们需要再次修改顶点结构体，为它添加一个纹理坐标以标识纹理上的点。此时，每个3D顶点就有了与之对应的2D纹理顶点。这样一来，用于定义3D三角形的3个顶点，也在纹理坐标系中定义了一个2D三角形（即我们要为每个3D三角形关联一个2D纹

理）。

```
struct Vertex
{
  DirectX::XMFLOAT3 Pos;
  DirectX::XMFLOAT3 Normal;
  DirectX::XMFLOAT2 TexC;
};

std::vector<D3D12_INPUT_ELEMENT_DESC> mInputLayout =
{
  { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
   D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
  { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
   D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
  { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
   D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
};
```

注意 Note

我们也可以创建与3D三角形在形状上有较大差异的2D纹理三角形来进行“特殊”的纹理贴图。如此一来，在2D纹理映射到3D三角形的过程中，所产生的拉伸与扭曲现象会令贴图的效果看上去不那么尽如人意。例如，将一个锐角三角形映射到一个直角三角形的时候就需要进行上述的拉伸操作。一般来讲，应尽量避免使纹理受到扭曲，除非贴图师（texture artist）希望获得拉伸的效果，故意而为之。

在图9.3中，我们将纹理图像一丝不差地映射到了立方体的每一个表面之上。然而这并不是唯一的选择：我们可以只向几何体映射部分纹

理。事实上，我们也能够将几个并无关联的图像合为一个大的纹理图（这被称为纹理图集，texture atlas），再将它应用于若干不同的物体（见图9.4）。此时，纹理坐标将用于确定纹理的哪一部分将被映射到目标三角形上。



图9.4　图中的纹理图集用一个大纹理存储了4种子纹理。此时，要通过为每个顶点所设置的纹理坐标来确定将要映射到目标几何体上的局部纹理

9.3 纹理数据源

贴图师通常会借助Photoshop或一些其他的图像编辑器为游戏制作纹理，最后再将它们保存为某种格式的图像文件，如BMP、DDS、TGA或PNG等。随后，游戏应用程序会在加载期间将图像文件载入`ID3D12Resource`对象。对于实时图形应用程序来说，DDS图像文件格式（DirectDraw图面格式，DirectDraw Surface format，DDS）是一种尚佳的选择：除了支持GPU可原生处理的各种图像格式，它还支持一些GPU自身就可解压的压缩图像格式。

注意 Note

贴图师们不宜将DDS格式当作工作过程中所用的图像格式，而是应当用他们认为更加顺手的格式来保存工作进程。待纹理最终完成后，再为游戏应用程序把它导出为DDS格式。

9.3.1 DDS格式概述

DDS对于3D图形来说是一种理想的格式，因为它支持一些专用于3D图形的特殊格式以及纹理类型。从本质上来讲，它实为一种针对GPU而专门设计的图像格式。例如，DDS纹理满足用于3D图形开发的

以下特征：

1．mipmap。

2．GPU能自行解压的压缩格式。

3．纹理数组。

4．立方体图（cube map，也有译作立方体贴图）。

5．体纹理（volume texture，也有译作体积纹理、立体纹理等）。

DDS格式能够支援不同的像素格式。像素格式由枚举类型DXGI_FORMAT中的成员来表示，但是并非所有的格式都适用于DDS纹理。非压缩图像数据一般会采用下列格式。

1．DXGI_FORMAT_B8G8R8A8_UNORM或DXGI_FORMAT_B8G8R8X8_UNORM：适用于低动态范围（low-dynamic-range）图像。

2．DXGI_FORMAT_R16G16B16A16_FLOAT：适用于高动态范围（high-dynamic-range）图像。

随着虚拟场景中纹理数量的大幅增长，对GPU端显存的需求也在迅速增加（还记得吗，我们需要将所有的纹理都置于显存当中，以便在程序中快速地运用这些资源）。为了缓解这些内存的需求压力，Direct3D支持下列几种压缩纹理格式（也称作块压缩，block compression）。

1. BC1（DXGI_FORMAT_BC1_UNORM）：如果我们需要将图片压缩为支持3个颜色通道和仅有1位（开/关）alpha分量的格式，则使用此格式。

2. BC2（DXGI_FORMAT_BC2_UNORM）：如果我们需要将图片压缩为支持3个颜色通道和仅有4位alpha分量的格式，则应用此格式。

3. BC3（DXGI_FORMAT_BC3_UNORM）：如果我们需要将图片压缩为支持3个颜色通道和8位alpha分量的格式，则采用此格式。

4. BC4（DXGI_FORMAT_BC4_UNORM）：如果我们需要将图片压缩为仅含有1个颜色通道的格式（如灰度图像），则运用此格式。

5. BC5（DXGI_FORMAT_BC5_UNORM）：如果我们需要将图片压缩为只支持2个颜色通道的格式，则使用此格式。

6. BC6（DXGI_FORMAT_BC6H_UF16）：如果我们需要将图片压缩为HDR（高动态范围）图像数据，则应用此格式。

7. BC7（DXGI_FORMAT_BC7_UNORM）：此格式用于对RGBA数据进行高质量的压缩。特别是，此格式可极大地减少因压缩法线图而造成的误差。

注意 Note

经压缩后的纹理只能用于输入到渲染流水线中的着色器阶段，而不能作为渲染目标。

注意 Note

由于块压缩算法（block compression algorithm）要以4 × 4的像素块为基础进行处理，所以纹理的尺寸必须为4的倍数。

再次重申，这些格式的优点是可以使图像以压缩的形式存于显存之中，而在需要时，GPU便能动态地对它们进行解压。将纹理压缩为DDS文件还有另一个好处，即更节省硬盘空间。

9.3.2　创建DDS文件

如果身为图形方面的编程新手，可能会对DDS比较陌生，因此会更多地使用如BMP、TGA或PNG这样的图像格式。下面介绍两种可以将传统图像格式转换为DDS格式的方法。

1．NVIDIA公司为Adobe Photoshop提供了一款可以将图像导出为DDS格式的插件。该插件现存于https://developer.nvidia.com/nvidia-texture-tools-adobe-photoshop。此插件还有一些其他选项，可供用户指

定DDS文件的`DXGI_FORMAT`格式，或生成mipmap等。

2．微软公司提供了一个名为texconv的命令行工具，该工具能将传统的图像格式转换为DDS文件。另外，*texconv*程序还有更多的其他功能，如调整图像大小、改变像素格式、生成mipmap等。可以在网站https://directxtex.codeplex.com/wikipage?title=Texconv&referringTitle=ocumentation找到它的文档与下载链接。

下面的示例展示了向*texconv*程序输入一个BMP文件bricks.bmp，并通过它来输出格式为`BC3_UNORM`且具有一个mipmap链（链中共有10个mipmap）的DDS文件bricks.dds。

```
texconv -m 10 -f BC3_UNORM bricks.bmp
```

注意 Note

微软公司还提供了另一个名为*texassemble*的命令行工具，该工具常被用于创建存有纹理数组、体纹理或立方体图的DDS文件。本书的后面会用到此工具[3]。

注意 Note

Visual Studio 2015中内置了一个支持DDS以及其他常见格式的内置图像编辑器。我们可以将图片拖入Visual Studio 2015，它便会自动用此编辑器将图像打开。对于DDS文件而言，我们可以在此编辑器中查看mipmap层级、修改DDS格式以及考察各种颜色通道等信息。

9.4 创建以及启用纹理

9.4.1 加载DDS文件

微软公司提供了一组用来加载DDS文件的轻量级源代码。

但是，在写下这段话的时候，此代码仅支持DirectX 11[4]。我们在此修改了DDSTextureLoader.h/cpp文件，专为DirectX 12提供了一个新加的读取DDS文件的方法（这两个已修改的文件可以在DVD或供读者下载的源文件中的Common文件夹内找到）。

```
HRESULT DirectX::CreateDDSTextureFromFile12(
  _In_ ID3D12Device* device,
  _In_ ID3D12GraphicsCommandList* cmdList,
  _In_z_ const wchar_t* szFileName,
  _Out_ Microsoft::WRL::ComPtr<ID3D12Resource>& texture,
  _Out_ Microsoft::WRL::ComPtr<ID3D12Resource>& textureUploadHeap);
```

1. `device`：指向用于创建纹理资源的D3D设备的指针。

2. `cmdList`：提交GPU命令（例如，将纹理数据从上传堆复制到默认堆的命令）的命令列表。

3. `szFileName`：欲加载的图像文件名。

4. `texture`：返回载有图像数据的纹理资源。

5. `textureUploadHeap`：返回的纹理资源，在此，将它当作一个

上传堆，用于将图像数据复制到默认堆中的纹理资源。在GPU完成其上述复制命令之前，不能销毁该资源。

为了用名为WoodCreate01.dds的图像来创建一个对应的纹理，应按照如下方式编写代码：

```
struct Texture
{
  // 为了便于查找而所用的唯一材质名
  std::string Name;

  std::wstring Filename;

  Microsoft::WRL::ComPtr<ID3D12Resource> Resource = nullptr;
  Microsoft::WRL::ComPtr<ID3D12Resource> UploadHeap = nullptr;
};

auto woodCrateTex = std::make_unique<Texture>();
woodCrateTex->Name = "woodCrateTex";
woodCrateTex->Filename = L"Textures/WoodCrate01.dds";
ThrowIfFailed(DirectX::CreateDDSTextureFromFile12(
  md3dDevice.Get(), mCommandList.Get(),
  woodCrateTex->Filename.c_str(),
  woodCrateTex->Resource, woodCrateTex->UploadHeap));
```

9.4.2 着色器资源视图堆

创建了纹理资源后，我们还需要为它再创建一个SRV（着色器资源视图）描述符，并将其设置到一个根签名参数槽（root signature parameter slot），以供着色器程序使用。为此，首先要用`ID3D12Device:: CreateDescriptorHeap`函数来创建描述符堆，借此存储SRV描述符。下面的代码构建了一个可容纳3个类型为CBV、SRV或UAV描述符的描述符堆，并使之在着色器中可见（即可供着色器

使用）：

```
D3D12_DESCRIPTOR_HEAP_DESC srvHeapDesc = {};
srvHeapDesc.NumDescriptors = 3;
srvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
srvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
ThrowIfFailed(md3dDevice->CreateDescriptorHeap(
  &srvHeapDesc, IID_PPV_ARGS(&mSrvDescriptorHeap)));
```

9.4.3 创建着色器资源视图描述符

一旦创建了SRV堆，便可创建真正的描述符。我们通过填写D3D12_SHADER_RESOURCE_VIEW_DESC对象来描述SRV描述符，该结构体详述了资源的类型以及其他的信息，如格式、维数、mipmap数量等。

```
typedef struct D3D12_SHADER_RESOURCE_VIEW_DESC
{
  DXGI_FORMAT Format;
  D3D12_SRV_DIMENSION ViewDimension;
  UINT Shader4ComponentMapping;
  union
  {
    D3D12_BUFFER_SRV Buffer;
    D3D12_TEX1D_SRV Texture1D;
    D3D12_TEX1D_ARRAY_SRV Texture1DArray;
    D3D12_TEX2D_SRV Texture2D;
    D3D12_TEX2D_ARRAY_SRV Texture2DArray;
    D3D12_TEX2DMS_SRV Texture2DMS;
    D3D12_TEX2DMS_ARRAY_SRV Texture2DMSArray;
    D3D12_TEX3D_SRV Texture3D;
    D3D12_TEXCUBE_SRV TextureCube;
    D3D12_TEXCUBE_ARRAY_SRV TextureCubeArray;
  };
} D3D12_SHADER_RESOURCE_VIEW_DESC;

typedef struct D3D12_TEX2D_SRV
{
```

```
  UINT MostDetailedMip;
  UINT MipLevels;
  UINT PlaneSlice;
  FLOAT ResourceMinLODClamp;
} D3D12_TEX2D_SRV;
```

对于2D纹理来说，我们只关心联合体中的D3D12_TEX2D_SRV部分。

1. Format：视图的格式。如果待创建视图的资源有具体的格式，即并非以无类型（typeless）的格式创建而成），就用此资源的DXGI_FORMAT格式来填写此参数。如果是通过无类型的DXGI_FORMAT来创建该资源的，则一定要在此为视图填写具体的类型，只有这样GPU才能知道怎样解释并处理这一数据。

2. ViewDimension：资源的维数。目前我们只使用2D纹理，所以将此参数指定为D3D12_SRV_DIMENSION_TEXTURE2D。以下是几种常见的纹理维数：

（a）D3D12_SRV_DIMENSION_TEXTURE1D：资源为1D纹理。

（b）D3D12_SRV_DIMENSION_TEXTURE3D：资源为3D纹理。

（c）D3D12_SRV_DIMENSION_TEXTURECUBE：资源为立方体纹理（cube texture）。

3. Shader4ComponentMapping：在着色器中对纹理进行采样时，它将返回特定纹理坐标处的纹理数据向量。这个字段提供了一种方法，可以将采样时所返回的纹理向量中的分量进行重新排序。例如，可以用

此字段将红色分量与绿色分量互换。该方法常用于一些特殊的场合，但本书中并不涉及这些情景。因此，只要将它指定为D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING即可。这样一来，向量分量的顺序将不会改变，它会以纹理资源中默认的数据顺序直接返回。

4. MostDetailedMip：指定此视图中图像细节最详尽的mipmap层级的索引。此参数的取值范围在0与MipLevels-1之间。

5. MipLevels：自MostDetailedMip算起，待创建视图的mipmap层级数量。通过将这个字段与MostDetailedMip配合起来，我们就能够指定此视图mipmap层级的连续子范围。可以将此字段设置为-1，用来表示自MostDetailedMip始至最后一个mipmap层级之间的所有mipmap级别。

6. PlaneSlice：平面切片的索引（详见12.3.4小节纹理子资源）。

7. ResourceMinLODClamp：指定可以访问的最小mipmap层级。设置为0.0表示可以访问所有的mipmap层级。将此参数指定为3.0，则表示可以访问从3.0到MipCount-1的mipmap层级。

接下来，让我们构建3个资源描述符来填充在上一小节中所创建的描述符堆。

```
// 假设已创建下列3个纹理资源
// ID3D12Resource* bricksTex;
```

```
// ID3D12Resource* stoneTex;
// ID3D12Resource* tileTex;

// 获取指向描述符堆起始处的指针
CD3DX12_CPU_DESCRIPTOR_HANDLE hDescriptor(
  mSrvDescriptorHeap->GetCPUDescriptorHandleForHeapStart());

D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING
;
srvDesc.Format = bricksTex->GetDesc().Format;
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Texture2D.MostDetailedMip = 0;
srvDesc.Texture2D.MipLevels = bricksTex->GetDesc().MipLevels;
srvDesc.Texture2D.ResourceMinLODClamp = 0.0f;
md3dDevice->CreateShaderResourceView(bricksTex.Get(), &srvDesc, hDescripto
r);

// 偏移到堆中的下一个描述符处
hDescriptor.Offset(1, mCbvSrvDescriptorSize);

srvDesc.Format = stoneTex->GetDesc().Format;
srvDesc.Texture2D.MipLevels = stoneTex->GetDesc().MipLevels;
md3dDevice->CreateShaderResourceView(stoneTex.Get(), &srvDesc, hDescriptor
);

// 偏移到堆中的下一个描述符处
hDescriptor.Offset(1, mCbvSrvDescriptorSize);

srvDesc.Format = tileTex->GetDesc().Format;
srvDesc.Texture2D.MipLevels = tileTex->GetDesc().MipLevels;
md3dDevice->CreateShaderResourceView(tileTex.Get(), &srvDesc, hDescriptor);
```

9.4.4 将纹理绑定到流水线

至此，我们在每次绘制调用时所指定的材质，都是由材质常量缓冲区来进行更新的。这就意味着在绘制调用的过程中，所有的几何体都将使用同一组材质数据。这对程序而言是一种极大的限制，因为我们将不能动态地指定每个像素的材质，继而导致场景细节的缺失。纹理映射技

术的想法是用纹理图（texture map）来取代材质常量缓冲区以获取材质数据。这将使每个像素的数据都是灵活可变化的，从而为场景增添更丰富的细节与几分真实感。

在本节中，我们将添加漫反射反照率纹理图（diffuse albedo texture map），以此来给出材质的漫反射反照率分量。影响材质的两个数值gFresnelR0与gRoughness仍将在每次绘制调用时由材质常量缓冲区来指定。而在第19章中，我们会介绍如何借助纹理在像素层级指定粗糙度。注意，在使用纹理贴图时，我们仍需在材质常量缓冲区中保留gDiffuseAlbedo分量。事实上，我们在像素着色器中会以下列方式令纹理漫反射反照率数据与DiffuseAlbedo相组合：

```
// 从纹理中提取此像素的漫反射反照率
float4 texDiffuseAlbedo = gDiffuseMap.Sample(
  gsamAnisotropicWrap, pin.TexC);

// 将纹理样本与常量缓冲区中的反照率相乘
float4 diffuseAlbedo = texDiffuseAlbedo * gDiffuseAlbedo;
```

我们通常设DiffuseAlbedo=(1,1,1,1)，从而使texDiffuseAlbedo不会发生改变。但是，有时对DiffuseAlbedo进行适当的调整却可以避免制作新的纹理，来看下这种情况：假设有一个砖块纹理，若贴图师希望使它的色调略显偏蓝，便可以通过设置DiffuseAlbedo=(0.9,0.9,1,1)削减其中的红色与绿色成分来达到这个目的。

我们向材质的定义中添加了一个索引，借此引用了与此材质相关联的纹理描述堆中的一个SRV：

```
struct Material
{
  ...

  // 漫反射纹理在SRV堆中的索引
  int DiffuseSrvHeapIndex = -1;

  ...
};
```

接下来，假设根签名被定义为需要把由着色器资源视图构成的描述符表绑定到第0个槽处，那么，我们便可以通过下列代码来使用纹理绘制渲染项：

```
void CrateApp::DrawRenderItems(
  ID3D12GraphicsCommandList* cmdList,
  const std::vector<RenderItem*>& ritems)
{
  UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(
    ObjectConstants));
  UINT matCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(
    MaterialConstants));

  auto objectCB = mCurrFrameResource->ObjectCB->Resource();
  auto matCB = mCurrFrameResource->MaterialCB->Resource();

  // 对于每个渲染项而言……
  for(size_t i = 0; i < ritems.size(); ++i)
  {
    auto ri = ritems[i];

    cmdList->IASetVertexBuffers(0, 1, &ri->Geo->VertexBufferView());
    cmdList->IASetIndexBuffer(&ri->Geo->IndexBufferView());
    cmdList->IASetPrimitiveTopology(ri->PrimitiveType);

    CD3DX12_GPU_DESCRIPTOR_HANDLE tex(
      mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());
    tex.Offset(ri->Mat->DiffuseSrvHeapIndex, mCbvSrvDescriptorSize);

    D3D12_GPU_VIRTUAL_ADDRESS objCBAddress =
      objectCB->GetGPUVirtualAddress() +
      ri->ObjCBIndex*objCBByteSize;
    D3D12_GPU_VIRTUAL_ADDRESS matCBAddress =
```

```
      matCB->GetGPUVirtualAddress() +
      ri->Mat->MatCBIndex*matCBByteSize;

    cmdList->SetGraphicsRootDescriptorTable(0, tex);
    cmdList->SetGraphicsRootConstantBufferView(1, objCBAddress);
    cmdList->SetGraphicsRootConstantBufferView(3, matCBAddress);

    cmdList->DrawIndexedInstanced(ri->IndexCount,
      1, ri->StartIndexLocation,
      ri->BaseVertexLocation, 0);
  }
}
```

注意 Note

实际上，纹理资源可以运用于任何着色器（如顶点着色器、几何着色器与像素着色器）。而我们暂时只将它应用于像素着色器。正如之前所述，纹理在本质上是种支持GPU特殊操作的特别数组，因此不难想象，它们在其他的着色器程序中也能发挥巨大的作用。

注意 Note

由于纹理图集可以在一次绘制调用中渲染出多个几何体，因此可以将它用于优化性能。比如说，假设我们使用了图8.4中含有板条箱、草地以及砖块等纹理的纹理图集。接着，通过将每个物体的纹理坐标调整至其相应的子纹理，我们就能将所有的几何体都放置在一个渲染项（假

设每个物体都没有其他需要修改的参数）中。通过与Direct3D的早期版本进行比较，我们可以发现Direct3D 12已大幅度降低了渲染过程中的开销。尽管如此，由于绘制调用的过程中依然会产生开销，因而仍需要这类技术，以期将绘制调用的次数降到最少。

9.5 过滤器[5]

9.5.1 放大

我们可以将纹理图中的元素看作是从连续图像中采集的离散颜色样本，但并不应认为它们是有着特定面积大小的矩形。所以，当前的疑问是：如果在纹理坐标(u, v)处没有与之对应的纹素点究竟会发生什么？这个问题有可能在下述情景中发生：假设玩家慢慢靠近了场景中的一堵墙壁，则墙壁将逐渐放大并占据完整的镜头。为了便于说明问题，这里假设显示器的分辨率为1024×1024，而且墙壁纹理的分辨率为256×256。由此就产生了纹理放大（magnification）的概念——我们试图用少量纹素来覆盖大量的像素。在上述示例中，每个纹素点之间都列有4个像素。当在三角形中对顶点纹理坐标进行插值时，每个像素都将得到其唯一的纹理坐标。如此一来，就有可能发生没有纹素点与纹理坐标处的像素相对应的情况（纹理与像素分辨率不匹配所造成）。对此，我们可以对纹素之间的颜色数据进行插值估算，从而获得指定纹素处的颜色信息。图形硬件往往会支持常数插值（constant interpolation，也有译作常量插值）与线性插值（linear interpolation）两种插值方法。在实践中，线性插值的使用更为普遍。

图9.5详细地描述了这两种方法在1D情况下的使用过程：假设我们有一个内含256个样本的1D纹理，并且某个插值纹理坐标为$u = 0.126484375$，所以此归一化纹理坐标就对应于

$0.126484375 \times 256 = 32.38$处的纹素。显而易见，此值实际位于两个纹素样本之间，所以我们必须通过插值这一手段来求取它的近似值。

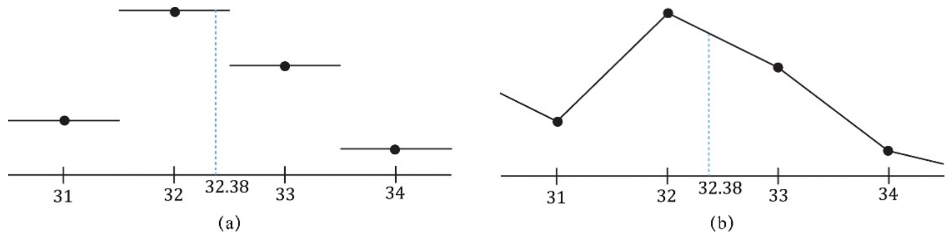


图9.5 图a中，若给出了纹素点，我们便可以通过构建分段常量函数来求出纹素点之间某处的近似值。由于得到的近似值为最近纹素点的取值，所以这种方法有时被称为最近邻点采样（nearest neighbor point sampling）。图b中，若给出了纹素点，我们便可以通过构建分段线性函数来求出纹素点之间某处的近似值

2D线性插值又称为双线性插值（bilinear interpolation），其处理流程如图9.6所示：给出四个纹素之间的一个纹理坐标，先在水平方向u上进行两次1D线性插值（求出$\mathbf{c}_T$与$\mathbf{c}_B$），后在垂直方向v上再进行一次1D内插（求取$\mathbf{c}$）。

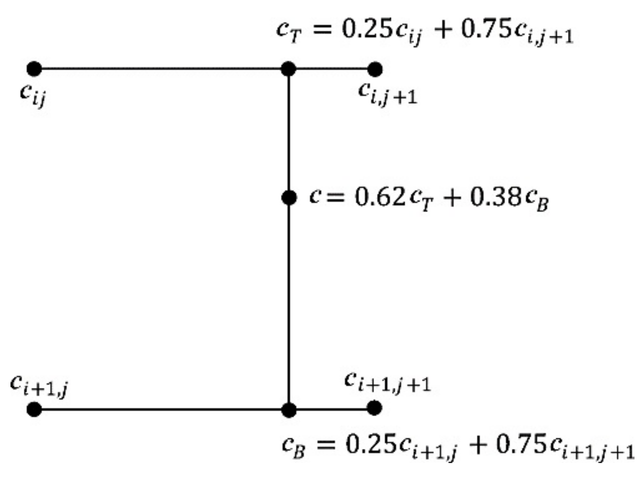


图9.6 图中给出了4个纹素点，分别为$\mathbf{c}_{i,j}$、$\mathbf{c}_{i,j+1}$、$\mathbf{c}_{i+1,j}$、$\mathbf{c}_{i+1,j+1}$。我们希望通过插值方法，近似地求出这4个纹素点之间的纹素点$\mathbf{c}$的颜色。在此例中，$\mathbf{c}$位于$\mathbf{c}_{i,j}$右侧的0.75单位处以及$\mathbf{c}_{i,j}$下侧的0.38单位处。我们首先对上侧两个纹素点的颜色进行插值，以求出$\mathbf{c}_T$。接着对下

侧

两个纹素点的颜色进行插值来求出c_B。最后，我们再对c_T与c_B进行1D线性插值来求取c

图9.7所示为常数插值与线性插值的差别。正如我们所看到的，用常数插值的方法会产生块状的模糊图像。线性插值的效果虽然相对更加平滑，但是经过插值所得到的数据仍不如真实数据（如更高分辨率的纹理）来得完美。

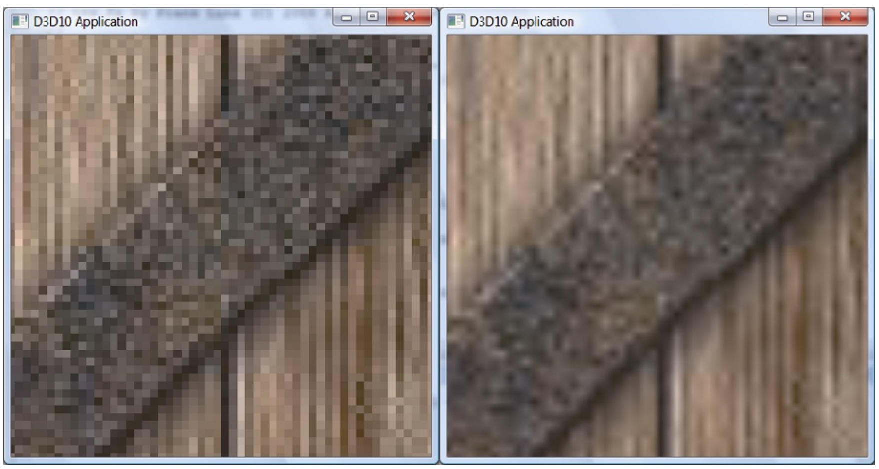


图9.7　由于玩家走近了具有板条箱纹理的立方体，因而触发了纹理放大操作。左图是采用常数插值所

得到的效果，看起来呈块状且模糊。出现这种结果也是合情合理的，因为采用的并不是具有连续性的插值

函数（见图9.5a），所以这将使纹素间颜色的变化比较突兀而不平滑。右图是利用线性过滤所得到的效果，由于此内插函数具有连续性，所以图像看起来更为平滑

还有一点需要注意的是，在以虚拟视角可以自由移动和探索的交互式3D程序中，纹理放大是个无法回避的问题。在与目标保持特定距离时，纹理可能看上去还不错，但是随着观察点逐渐接近目标，其效果就开始惨不忍睹了。一些游戏对虚拟视角过于接近物体表面的行为进行了限制，以避免过度的放大处理。通过使用更高分辨率的纹理也可以使此

难题得到一定程度的改善。

注意 Note

在纹理这一语境中，通过常数插值来求得纹素之间纹理坐标处的纹理数据也称为点过滤（point filtering）。为了求取纹素之间纹理坐标处的纹理数据而使用线性插值的计算方法，也称为线性过滤（linear flltering）。点过滤与线性过滤是Direct3D中所常用的术语。

9.5.2　缩小

纹理缩小（minification）是纹理放大的逆运算。在缩小的过程中，大量纹素将被映射到少数纹理之上。例如，考虑下列情景：假设有一堵被256×256纹理所映射的墙壁，玩家的观察视角正紧盯着它，并逐渐向后退却。在此过程中，这堵墙会看上去越来越小，直至它在屏幕上只覆盖大小为64×64像素的区域。此时，我们就应当将256×256纹素映射到64×64屏幕像素。在这种情况下，像素的纹理坐标处往往不会有与之对应的纹理图纹素，因此还需要将常数插值过滤器与线性插值过滤器运用于纹理缩小的情形。然而，在执行纹理缩小操作时还有更多的工作要做。从直观上来讲，通过平均下采样（average downsampling）应当可以使256×256纹素减少到64×64纹素。而mipmap技术则以占用一些额外

的内存为代价来实现与之相似的功能。在初始化期间（或资源创建时期），通过对图像下采样来创建mipmap链便可制作出缩小版的纹理（见图9.8）[6]。因此，这里所指的求平均值所做的工作实际上就是针对mipmap的大小执行预计算（提前制作出不同规格的纹理）。在运行时，图形硬件将根据程序员的设定，从以下两种不同的执行方案中择一而行：

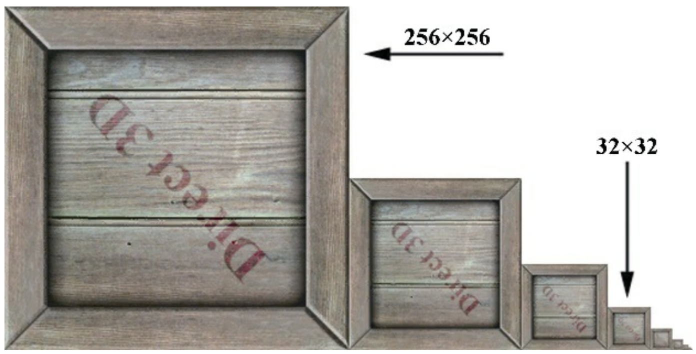


图9.8　图中所示的是一个mipmap链。每对相邻的mipmap中，后者都是前者大小的1/4。mipmap的大小最小可至1×1

1．在纹理贴图时，选择与待投影到屏幕上的几何体分辨率最为匹配的mipmap层级，并根据具体需求选用常数插值或线性插值。这便是针对**mipmap**的点过滤（point filtering），该名称的由来是因为此种方法与常数插值很相似——我们仅选取与目标分辨率最邻近的那个mipmap层级并用它进行纹理贴图。

2．在纹理贴图时，选取与待投影到屏幕上的几何体分辨率最为匹配的两个邻近的mipmap层级（一个稍大于屏幕上几何体的分辨率，一个稍小于屏幕上几何体的分辨率）。接下来，对这两种mipmap层级分别应用常量过滤或线性过滤，以生成它们各自相应的纹理颜色。最后，

在这两种插值纹理之间再次进行颜色的插值计算。这个过程称为**mipmap**的线性过滤（linear filtering），原因是这种方法与线性插值比较相似——我们需要对目标分辨率最邻近的两个mipmap层级进行插值计算。

通过从mipmap链中选取恰当的纹理细节级别，可大大减少纹理缩小的运算次数。

注意 Note

如9.3.2节中所述，可用Photoshop DDS格式导出插件或使用texconv程序来创建mipmap。这些程序基于原始的图像数据，运用下采样算法来生成更低的mipmap层级图像。有时候这些算法并不能保留所希望的图像细节，因此还要请贴图师亲手创建或编辑更低mipmap级别的图像，以保证不流失重要的细节。

9.5.3　各向异性过滤

还有一种名为各向异性过滤（anisotropic filtering）的过滤器类型。该过滤器有助于缓解当多边形法向量与摄像机观察向量之间夹角过大（比如当多边形正交于观察窗口时）所导致的失真现象。这种过滤器的开销最大，但是其校正失真的效果的确对得起它所消耗的资源。图9.9

展示了各向异性过滤与线性过滤两者的比较效果。



图9.9　板条箱的顶面基本上已正交于观察窗口。左图中的板条箱顶面采用了线性过滤，其效果模糊得一塌糊涂。右图以同样的角度观察通过各向异性过滤绘制的板条箱顶面，却呈现出细节更佳的渲染效果

9.6　寻址模式

可将经过常数插值或线性插值的纹理定义为一个返回向量值的函数$\boldsymbol{T}(u,v)=(r,g,b,a)$，即给定纹理坐标$(u,v)\in[0,\ 1]^2$，则上述纹理函数$\boldsymbol{T}$将返回颜色$(r,g,b,a)$。Direct3D允许我们采用下列4种不同方式（即寻址模式，address mode）来扩充此函数的定义域（解决输入值超出定义域这一问题），它们是重复寻址模式（wrap）、边框颜色寻址模式（border color，也有译作边界颜色寻址模式）、钳位寻址模式（clamp）与镜像寻址模式（mirror）。

1．重复寻址模式通过在坐标的每个整数点（integer junction）处重复绘制图像来拓充纹理函数（见图9.10）。

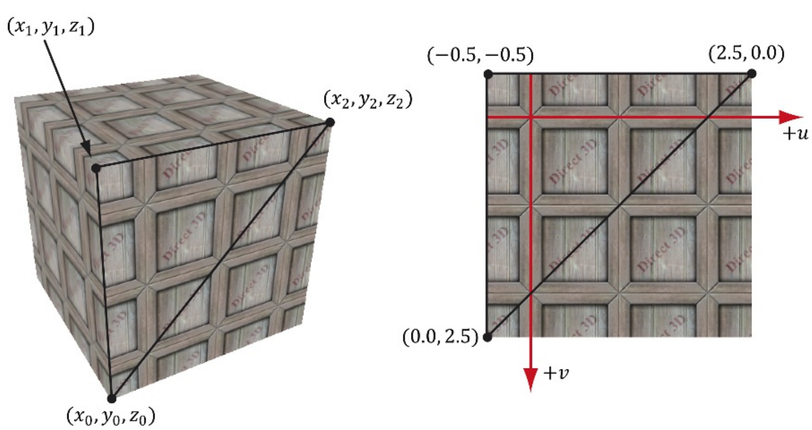


图9.10　重复寻址模式

2．边框颜色寻址模式通过将每个不在范围$[0,\ 1]^2$内的坐标(u,v)都映射为程序员指定的颜色而拓充纹理函数（见图9.11）。

3．钳位寻址模式通过将范围$[0,\ 1]^2$外的每个坐标(u, v)都映射为颜色$\boldsymbol{T}(u_0, v_0)$来扩充纹理函数，其中，(u_0, v_0)为范围$[0,\ 1]^2$内距离(u, v)最近的点（见图9.12）。

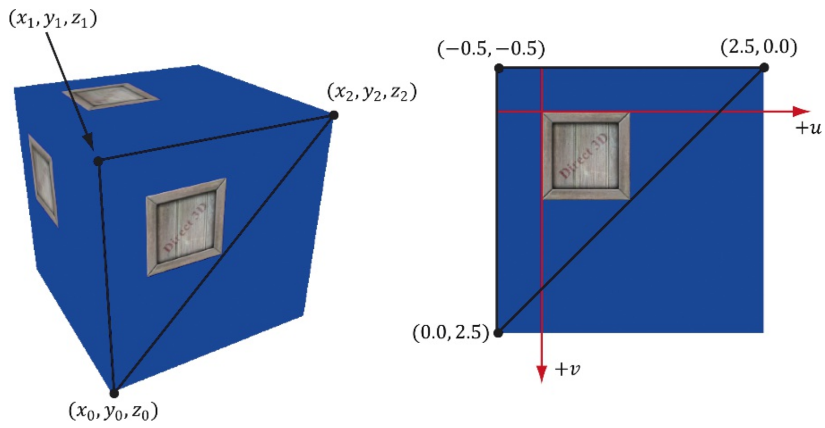


图9.11　边框颜色寻址模式

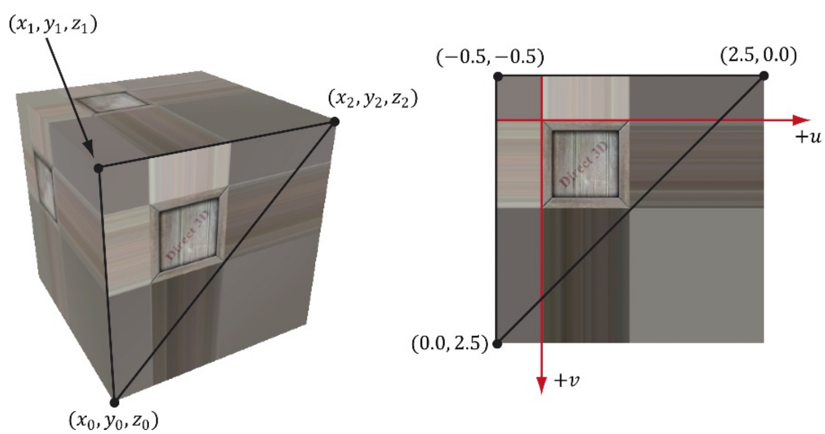


图9.12　钳位寻址模式

4．镜像寻址模式通过在坐标的每个整数点处绘制图像的镜像来扩充纹理函数（见图9.13）。

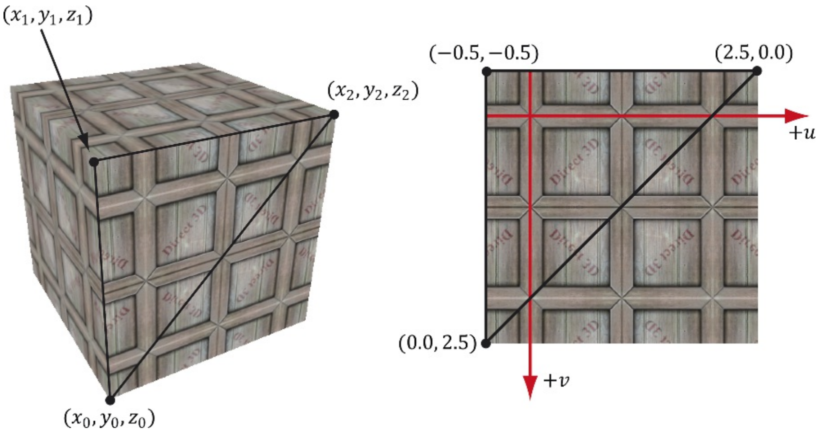


图9.13　镜像寻址模式

在程序中总是要指定一种寻址模式的（默认为重复寻址模式），因此在范围[0, 1]之外的纹理坐标也必然有定义。

重复寻址也许是最常使用的一种模式，它允许我们将一种纹理反复平铺到某一表面上。也就是说，这种寻址模式使我们在不提供额外数据的情况下，亦可提升纹理的分辨率（尽管范围$[0,\ 1]^2$之外的内容是重复的）。在执行平铺的时候，纹理是否能无缝衔接往往也是一个重点。例如，若板条箱纹理原本并不是无缝连接的，那么我们就能够明显地看出它是在反复贴同一张图。然而，图9.14中所展示的可无缝衔接砖块纹理却重复绘制了2×3次，读者是否对此有所察觉呢？

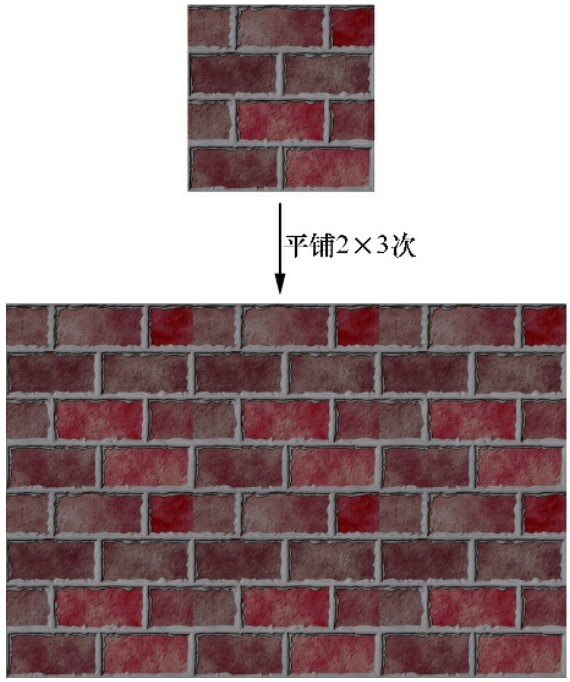


图9.14　将一种砖块纹理平铺2×3次的效果。由于此纹理是一种无缝贴图，因此很难察觉到我们采用了重复寻址模式

在Direct3D中，寻址模式由枚举类型D3D12_TEXTURE_ADDRESS_MODE来表示：

```
typedef enum D3D12_TEXTURE_ADDRESS_MODE
{
  D3D12_TEXTURE_ADDRESS_MODE_WRAP        = 1,
  D3D12_TEXTURE_ADDRESS_MODE_MIRROR      = 2,
  D3D12_TEXTURE_ADDRESS_MODE_CLAMP       = 3,
  D3D12_TEXTURE_ADDRESS_MODE_BORDER      = 4,
  D3D12_TEXTURE_ADDRESS_MODE_MIRROR_ONCE = 5
} D3D12_TEXTURE_ADDRESS_MODE;
```

9.7 采样器对象

从前面两个小节中可以看出，在运用纹理的过程中，除了纹理数据本身之外，还有另外两个相关的重要概念，即纹理过滤以及寻址模式。采集纹理资源时所用的过滤器和寻址模式都是由采样器对象（sampler object）来定义的。一个应用程序通常需要采用若干个采样器对象以不同的方式来采集纹理。

9.7.1 创建采样器

正如我们将在下一小节中所看到的，采样器会被着色器所用。为了将采样器绑定到着色器上供其使用，我们就需要为采样器对象绑定描述符。下面的代码展示了这样的一个根签名示例，它的第二个槽位获取了一个描述符表，此表中存有1个与采样器寄存器槽0相绑定的采样器描述符。

```
CD3DX12_DESCRIPTOR_RANGE descRange[3];
descRange[0].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1, 0);
descRange[1].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SAMPLER, 1, 0);
descRange[2].Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 0);

CD3DX12_ROOT_PARAMETER rootParameters[3];
rootParameters[0].InitAsDescriptorTable(1, &descRange[0],
    D3D12_SHADER_VISIBILITY_PIXEL);
rootParameters[1].InitAsDescriptorTable(1, &descRange[1],
    D3D12_SHADER_VISIBILITY_PIXEL);
rootParameters[2].InitAsDescriptorTable(1, &descRange[2],
    D3D12_SHADER_VISIBILITY_ALL);

CD3DX12_ROOT_SIGNATURE_DESC descRootSignature;
descRootSignature.Init(3, rootParameters, 0, nullptr,
```

```
  D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
```

如果要设置采样器描述符，还需一个采样器堆。而要创建采样器堆，就应通过填写D3D12_DESCRIPTOR_HEAP_DESC结构体实例并将其堆类型指定为D3D12_DESCRIPTOR_HEAP_TYPE_SAMPLER：

```
D3D12_DESCRIPTOR_HEAP_DESC descHeapSampler = {};
descHeapSampler.NumDescriptors = 1;
descHeapSampler.Type = D3D12_DESCRIPTOR_HEAP_TYPE_SAMPLER;
descHeapSampler.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;

ComPtr<ID3D12DescriptorHeap> mSamplerDescriptorHeap;
ThrowIfFailed(mDevice->CreateDescriptorHeap(&descHeapSampler,
  __uuidof(ID3D12DescriptorHeap),
  (void**)&mSamplerDescriptorHeap));
```

只要有了采样器堆，就能创建采样器描述符了。此时，我们再通过填写D3D12_SAMPLER_DESC对象来指定寻址模式、过滤器类型以及其他一些参数。

```
typedef struct D3D12_SAMPLER_DESC
{
  D3D12_FILTER Filter;
  D3D12_TEXTURE_ADDRESS_MODE AddressU;
  D3D12_TEXTURE_ADDRESS_MODE AddressV;
  D3D12_TEXTURE_ADDRESS_MODE AddressW;
  FLOAT MipLODBias;
  UINT MaxAnisotropy;
  D3D12_COMPARISON_FUNC ComparisonFunc;
  FLOAT BorderColor[ 4 ];
  FLOAT MinLOD;
  FLOAT MaxLOD;
} D3D12_SAMPLER_DESC;
```

1. Filter：D3D12_FILTER枚举类型的成员之一，用以指定采样纹理时所用的过滤方式。

2. AddressU：纹理在水平u轴方向上所用的寻址模式。

3. AddressV：纹理在垂直v轴方向上所用的寻址模式。

4. AddressW：纹理在深度w轴方向上所用的寻址模式（仅限于3D纹理）。

5. MipLODBias：设置mipmap层级的偏置值。如果将此值指定为0.0，则表示mipmap层级保持不变；若将此参数设置为2，mipmap层级设置为3，则将按层级5（3+2）进行采样。

6. MaxAnisotropy：最大各向异性值，此参数的取值区间为[1,16]。只有将Filter设置为D3D12_FILTER_ANISOTROPIC或D3D12_FILTER_COMPARISON_ANISOTROPIC之后该项才能生效。此数值越大也就意味着开销越大，但是用户同时会获得更棒的渲染效果。

7. ComparisonFunc：用于实现像阴影贴图（shadow mapping）这样一类特殊应用的高级选项。在未触及阴影贴图等章节之前，暂时只将它设置为D3D12_COMPARISON_FUNC_ALWAYS。

8. BorderColor：用于指定在D3D12_TEXTURE_ADDRESS_MODE_BORDER寻址模式下的边框颜色。

9. MinLOD：可供选择的最小mipmap层级。

10. MaxLOD：可供选择的最大mipmap层级。

以下是一些D3D12_FILTER类型的常用选项。

1．D3D12_FILTER_MIN_MAG_MIP_POINT：对纹理图与mipmap层级（即最接近于目标物体分辨率的mipmap层级）进行点过滤。

2．D3D12_FILTER_MIN_MAG_LINEAR_MIP_POINT：对纹理图进行双线性过滤，但对mipmap层级（即最接近于目标物体分辨率的mipmap层级）使用点过滤。

3．D3D12_FILTER_MIN_MAG_MIP_LINEAR：对纹理图进行双线性过滤，还要在两个邻近的较高、较低mipmap层级之间进行线性过滤。因此，这也被称为三线性过滤（trilinear filtering）。

4．D3D12_FILTER_ANISOTROPIC：对于纹理的放大、缩小以及mipmap均采用向异性过滤。

我们可以根据这组示例推测出其他可能的组合选项，也可以通过查阅SDK文档来了解D3D12_FILTER中的其他枚举类型。

以下实例展示了如何在描述符堆中为采样器创建出对应的描述符，该采样器使用的是线性过滤与重复寻址模式，其他参数则保留其默认值：

```
D3D12_SAMPLER_DESC samplerDesc = {};
samplerDesc.Filter = D3D12_FILTER_MIN_MAG_MIP_LINEAR;
samplerDesc.AddressU = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.AddressV = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.AddressW = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.MinLOD = 0;
samplerDesc.MaxLOD = D3D12_FLOAT32_MAX;
```

```
samplerDesc.MipLODBias = 0.0f;
samplerDesc.MaxAnisotropy = 1;
samplerDesc.ComparisonFunc = D3D12_COMPARISON_FUNC_ALWAYS;

md3dDevice->CreateSampler(&samplerDesc,
  mSamplerDescriptorHeap->GetCPUDescriptorHandleForHeapStart());
```

下面的代码则说明了怎样将采样器描述符绑定到预定的根签名参数槽，以供着色器程序使用：

```
commandList->SetGraphicsRootDescriptorTable(1,
  mSamplerDescriptorHeap->GetGPUDescriptorHandleForHeapStart());
```

9.7.2 静态采样器

事实证明，图形应用程序通常不会使用过多的采样器。为此，Direct3D专门提供了一种特殊的方式来定义采样器数组，使用户可以在不创建采样器堆的情况下也能对它们进行配置。CD3DX12_ROOT_SIGNATURE_DESC类有两种参数不同的Init函数，用户可以借此为应用程序定义所用的静态采样器数组。我们通过结构体D3D12_STATIC_SAMPLER_DESC来描述静态采样器，它与D3D12_SAMPLER_DESC结构体比较相似，但在以下方面存在区别。

1．边框颜色存在一些限制，即静态采样器的边框颜色必须为下列成员之一。

```
enum D3D12_STATIC_BORDER_COLOR
{
  D3D12_STATIC_BORDER_COLOR_TRANSPARENT_BLACK = 0,
  D3D12_STATIC_BORDER_COLOR_OPAQUE_BLACK = (
    D3D12_STATIC_BORDER_COLOR_TRANSPARENT_BLACK + 1 ) ,
  D3D12_STATIC_BORDER_COLOR_OPAQUE_WHITE = (
```

```
    D3D12_STATIC_BORDER_COLOR_OPAQUE_BLACK + 1 )
} D3D12_STATIC_BORDER_COLOR;
```

2．含有额外的字段用来指定着色器寄存器、寄存器空间以及着色器的可见性，这些其实都是配置采样器堆的相关参数。另外，用户只能定义2032个静态采样器，对于大多数应用程序而言是够用了。然而，如果这个数量真的无法使我们满足，那就只有另建采样器堆来使用非静态采样器了。

我们在演示程序中所用的是静态采样器，下述代码展示了怎样来定义它们。注意，我们的演示程序可能并不会用到所有的这些静态采样器，但却会一直保留这些定义，这样一来，在需要的时候便触手可及。这些预留的静态采样器并不多，而且它们也不会影响后续再定义的或用或不用的其他采样器。

```
std::array<const CD3DX12_STATIC_SAMPLER_DESC, 6>
  TexColumnsApp::GetStaticSamplers()
{
  // 应用程序一般只会用到这些采样器中的一部分
  // 所以就将它们全部提前定义好，并作为根签名的一部分保留下来

  const CD3DX12_STATIC_SAMPLER_DESC pointWrap(
    0, // 着色器寄存器
    D3D12_FILTER_MIN_MAG_MIP_POINT, // 过滤器类型
    D3D12_TEXTURE_ADDRESS_MODE_WRAP, // U轴方向上所用的寻址模式
    D3D12_TEXTURE_ADDRESS_MODE_WRAP, // V轴方向上所用的寻址模式
    D3D12_TEXTURE_ADDRESS_MODE_WRAP); // W轴方向上所用的寻址模式

  const CD3DX12_STATIC_SAMPLER_DESC pointClamp(
    1, // 着色器寄存器
    D3D12_FILTER_MIN_MAG_MIP_POINT, // 过滤器类型
    D3D12_TEXTURE_ADDRESS_MODE_CLAMP, // U轴方向上所用的寻址模式
    D3D12_TEXTURE_ADDRESS_MODE_CLAMP, // V轴方向上所用的寻址模式
    D3D12_TEXTURE_ADDRESS_MODE_CLAMP); // W轴方向上所用的寻址模式

  const CD3DX12_STATIC_SAMPLER_DESC linearWrap(
    2, // 着色器寄存器
```

```
    D3D12_FILTER_MIN_MAG_MIP_LINEAR, // 过滤器类型
    D3D12_TEXTURE_ADDRESS_MODE_WRAP, // U轴方向上所用的寻址模式
    D3D12_TEXTURE_ADDRESS_MODE_WRAP, // V轴方向上所用的寻址模式
    D3D12_TEXTURE_ADDRESS_MODE_WRAP); // W轴方向上所用的寻址模式

  const CD3DX12_STATIC_SAMPLER_DESC linearClamp(
    3, // 着色器寄存器
    D3D12_FILTER_MIN_MAG_MIP_LINEAR, // 过滤器类型
    D3D12_TEXTURE_ADDRESS_MODE_CLAMP, // U轴方向上所用的寻址模式
    D3D12_TEXTURE_ADDRESS_MODE_CLAMP, // V轴方向上所用的寻址模式
    D3D12_TEXTURE_ADDRESS_MODE_CLAMP); // W轴方向上所用的寻址模式

  const CD3DX12_STATIC_SAMPLER_DESC anisotropicWrap(
    4, // 着色器寄存器
    D3D12_FILTER_ANISOTROPIC, // 过滤器类型
    D3D12_TEXTURE_ADDRESS_MODE_WRAP, // U轴方向上所用的寻址模式
    D3D12_TEXTURE_ADDRESS_MODE_WRAP, // V轴方向上所用的寻址模式
    D3D12_TEXTURE_ADDRESS_MODE_WRAP, // W轴方向上所用的寻址模式
    0.0f,                // mipmap层级的偏置值
    8);                 // 最大各向异性值

  const CD3DX12_STATIC_SAMPLER_DESC anisotropicClamp(
    5, // 着色器寄存器
    D3D12_FILTER_ANISOTROPIC, // 过滤器类型
    D3D12_TEXTURE_ADDRESS_MODE_CLAMP, // U轴方向上所用的寻址模式
    D3D12_TEXTURE_ADDRESS_MODE_CLAMP, // V轴方向上所用的寻址模式
    D3D12_TEXTURE_ADDRESS_MODE_CLAMP, // W轴方向上所用的寻址模式
    0.0f,                // mipmap层级的偏置值
    8);                  // 最大各向异性值

  return {
    pointWrap, pointClamp,
    linearWrap, linearClamp,
    anisotropicWrap, anisotropicClamp };
}

void TexColumnsApp::BuildRootSignature()
{
  CD3DX12_DESCRIPTOR_RANGE texTable;
  texTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1, 0);

  // 根参数可以是描述符表、根描述符或根常量
  CD3DX12_ROOT_PARAMETER slotRootParameter[4];

  slotRootParameter[0].InitAsDescriptorTable(1,
    &texTable, D3D12_SHADER_VISIBILITY_PIXEL);
  slotRootParameter[1].InitAsConstantBufferView(0);
```

```
  slotRootParameter[2].InitAsConstantBufferView(1);
  slotRootParameter[3].InitAsConstantBufferView(2);

  auto staticSamplers = GetStaticSamplers();

  // 根签名即是一系列根参数
  CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(4, slotRootParameter,
    (UINT)staticSamplers.size(), staticSamplers.data(),
    D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

  // 创建具有4个槽位的根签名，第一个指向含有单个着色器资源视图的描述符表，其他3个
各指向一个常
  // 量缓冲区视图
  ComPtr<ID3DBlob> serializedRootSig = nullptr;
  ComPtr<ID3DBlob> errorBlob = nullptr;
  HRESULT hr = D3D12SerializeRootSignature(&rootSigDesc,
    D3D_ROOT_SIGNATURE_VERSION_1,
    serializedRootSig.GetAddressOf(), errorBlob.GetAddressOf());

  if(errorBlob != nullptr)
  {
    ::OutputDebugStringA((char*)errorBlob->GetBufferPointer());
  }
  ThrowIfFailed(hr);

  ThrowIfFailed(md3dDevice->CreateRootSignature(
    0,
    serializedRootSig->GetBufferPointer(),
    serializedRootSig->GetBufferSize(),
    IID_PPV_ARGS(mRootSignature.GetAddressOf())));
}
```

9.8　在着色器中对纹理进行采样

通过下列HLSL语法来定义纹理对象，并将其分配给特定的纹理寄存器：

```
Texture2D gDiffuseMap : register(t0);
```

注意，纹理寄存器由tn来标定，其中，整数n表示的是纹理寄存器的槽号。此根签名的定义指出了由槽位参数到着色器寄存器的映射关系，这便是应用程序代码能将SRV绑定到着色器中特定Texture2D对象的原因。

类似地，下列HLSL语法定义了多个采样器对象，并将它们分别分配到了特定的采样器寄存器。

```
SamplerState gsamPointWrap    : register(s0);
SamplerState gsamPointClamp    : register(s1);
SamplerState gsamLinearWrap    : register(s2);
SamplerState gsamLinearClamp   : register(s3);
SamplerState gsamAnisotropicWrap : register(s4);
SamplerState gsamAnisotropicClamp : register(s5);
```

这些采样器对应于我们在上一节中所配置的静态采样器数组。注意，采样器寄存器由sn来指定，其中整数n表示的是采样器寄存器的槽号。

现在，我们在像素着色器中为每个像素都指定其相应的纹理坐标(u, v)，并通过Texture2D::Sample方法正式地进行纹理采样。

```
Texture2D gDiffuseMap : register(t0);

SamplerState gsamPointWrap    : register(s0);
SamplerState gsamPointClamp    : register(s1);
SamplerState gsamLinearWrap    : register(s2);
SamplerState gsamLinearClamp   : register(s3);
SamplerState gsamAnisotropicWrap : register(s4);
SamplerState gsamAnisotropicClamp : register(s5);

struct VertexOut
{
  float4 PosH  : SV_POSITION;
  float3 PosW  : POSITION;
  float3 NormalW : NORMAL;
  float2 TexC  : TEXCOORD;
};

float4 PS(VertexOut pin) : SV_Target
{
  float4 diffuseAlbedo = gDiffuseMap.Sample(gsamAnisotropicWrap, pin.TexC)

    * gDiffuseAlbedo;
  ...
```

我们通过向Sample方法的第一个参数传递SamplerState对象来描述如何对纹理数据进行采样，再向第二个参数传递像素的纹理坐标(u, v)。这个方法将利用SamplerState对象所指定的过滤方法，返回纹理图在点(u, v)处的插值颜色。

9.9 板条箱演示程序

我们现在开始学习向立方体添加板条箱纹理的关键步骤（见图9.1）。

9.9.1 指定纹理坐标

`GeometryGenerator::CreateBox`函数用于生成立方体的纹理坐标，因此得以使纹理图像映射到立方体的每个表面。考虑到篇幅原因，这里只给出立方体前表面、后表面以及上表面的顶点定义。另外，我们还省略了`Vertex`构造函数中法线以及切向量的坐标（纹理坐标的有关部分字体已被加粗）。

```
GeometryGenerator::MeshData GeometryGenerator::CreateBox(
  float width, float height, float depth,
  uint32 numSubdivisions)
{
  MeshData meshData;

  Vertex v[24];

  float w2 = 0.5f*width;
  float h2 = 0.5f*height;
  float d2 = 0.5f*depth;

  // 填写立方体前表面的顶点数据
  v[0] = Vertex(-w2, -h2, -d2, ..., 0.0f, 1.0f);
  v[1] = Vertex(-w2, +h2, -d2, ..., 0.0f, 0.0f);
  v[2] = Vertex(+w2, +h2, -d2, ..., 1.0f, 0.0f);
  v[3] = Vertex(+w2, -h2, -d2, ..., 1.0f, 1.0f);

  // 填写立方体后表面的顶点数据
  v[4] = Vertex(-w2, -h2, +d2, ..., 1.0f, 1.0f);
  v[5] = Vertex(+w2, -h2, +d2, ..., 0.0f, 1.0f);
```

```
  v[6] = Vertex(+w2, +h2, +d2, ..., 0.0f, 0.0f);
  v[7] = Vertex(-w2, +h2, +d2, ..., 1.0f, 0.0f);

  // 填写立方体上表面的顶点数据
  v[8]  = Vertex(-w2, +h2, -d2, ..., 0.0f, 1.0f);
  v[9]  = Vertex(-w2, +h2, +d2, ..., 0.0f, 0.0f);
  v[10] = Vertex(+w2, +h2, +d2, ..., 1.0f, 0.0f);
  v[11] = Vertex(+w2, +h2, -d2, ..., 1.0f, 1.0f);
  ...
```

如果忘记了纹理坐标为何如此定义，可回顾图9.3。

9.9.2 创建纹理

我们通过下列代码在初始化阶段利用dds文件来创建纹理。

```
// 将纹理相关数据组织在一起的辅助结构体
struct Texture
{
  // 便于查找材质所用的唯一名称
  std::string Name;

  std::wstring Filename;

  Microsoft::WRL::ComPtr<ID3D12Resource> Resource = nullptr;
  Microsoft::WRL::ComPtr<ID3D12Resource> UploadHeap = nullptr;
};

std::unordered_map<std::string, std::unique_ptr<Texture>> mTextures;

void CrateApp::LoadTextures()
{
  auto woodCrateTex = std::make_unique<Texture>();
  woodCrateTex->Name = "woodCrateTex";
  woodCrateTex->Filename = L"Textures/WoodCrate01.dds";
  ThrowIfFailed(DirectX::CreateDDSTextureFromFile12(md3dDevice.Get(),
    mCommandList.Get(), woodCrateTex->Filename.c_str(),
    woodCrateTex->Resource, woodCrateTex->UploadHeap));

  mTextures[woodCrateTex->Name] = std::move(woodCrateTex);
}
```

我们将每个彼此独立的纹理都存于一个无序映射表（unordered map）之中，再根据它们各自的名称来查找相应的纹理。在实际的产品代码中，我们应当在纹理加载之前检测它的数据是否已经存在（即它是否已经被加载于无序映射表之中），以防发生同一纹理被加载多次的情况。

9.9.3 设置纹理

如果纹理已被创建，并且它的SRV也存在于描述符堆中，那么，我们只要把所需纹理设置到根签名参数以将其绑定至渲染流水线，便能使它在着色器程序中得以使用。

```
// 获取欲绑定纹理的SRV
CD3DX12_GPU_DESCRIPTOR_HANDLE tex(
mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());
tex.Offset(ri->Mat->DiffuseSrvHeapIndex, mCbvSrvDescriptorSize);

...

// 将纹理SRV绑定到根参数0。根参数指示了该纹理将要具体绑定到哪一个着色器寄存器槽
cmdList->SetGraphicsRootDescriptorTable(0, tex);
```

9.9.4 更新HLSL部分代码

以下是经过修改的Default.hlsl文件，现已支持纹理贴图（与纹理贴图有关的代码部分为粗体字）。

```
// 默认的光源数量
#ifndef NUM_DIR_LIGHTS
  #define NUM_DIR_LIGHTS 3
```

```
#endif

#ifndef NUM_POINT_LIGHTS
  #define NUM_POINT_LIGHTS 0
#endif

#ifndef NUM_SPOT_LIGHTS
  #define NUM_SPOT_LIGHTS 0
#endif

// 包含了光照所用的结构体和函数
#include "LightingUtil.hlsl"

Texture2D  gDiffuseMap : register(t0);

SamplerState gsamPointWrap        : register(s0);
SamplerState gsamPointClamp       : register(s1);
SamplerState gsamLinearWrap       : register(s2);
SamplerState gsamLinearClamp      : register(s3);
SamplerState gsamAnisotropicWrap  : register(s4);
SamplerState gsamAnisotropicClamp : register(s5);

// 每一帧都有变化的常量数据
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorld;
  float4x4 gTexTransform;
};

// 绘制过程中所用的杂项常量数据
cbuffer cbPass : register(b1)
{
  float4x4 gView;
  float4x4 gInvView;
  float4x4 gProj;
  float4x4 gInvProj;
  float4x4 gViewProj;
  float4x4 gInvViewProj;
  float3 gEyePosW;
  float cbPerObjectPad1;
  float2 gRenderTargetSize;
  float2 gInvRenderTargetSize;
  float gNearZ;
  float gFarZ;
  float gTotalTime;
  float gDeltaTime;
  float4 gAmbientLight;
```

```
  // 对于每个以MaxLights为光源数量最大值的对象而言，索引[0,NUM_DIR_LIGHTS)表示的是方向光
  // 源，索引[NUM_DIR_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHTS)表示的是点光源
  // 索引[NUM_DIR_LIGHTS+NUM_POINT_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHT+NUM_SPOT_ LIGHTS)表示的是聚光灯光源
  Light gLights[MaxLights];
};

// 每种材质都有所区别的常量数据
cbuffer cbMaterial : register(b2)
{
  float4  gDiffuseAlbedo;
  float3  gFresnelR0;
  float  gRoughness;
  float4x4 gMatTransform;
};

struct VertexIn
{
  float3 PosL  : POSITION;
  float3 NormalL : NORMAL;
  float2 TexC  : TEXCOORD;
};

struct VertexOut
{
  float4 PosH  : SV_POSITION;
  float3 PosW  : POSITION;
  float3 NormalW : NORMAL;
  float2 TexC  : TEXCOORD;
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout = (VertexOut)0.0f;

  // 把坐标变换到世界空间
  float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
  vout.PosW = posW.xyz;

  // 假设这里正在进行的是等比缩放，否则便需要使用世界矩阵的逆转置矩阵
  vout.NormalW = mul(vin.NormalL, (float3x3)gWorld);

  // 将顶点变换到齐次裁剪空间
  vout.PosH = mul(posW, gViewProj);
```

```
  // 为了对三角形进行插值操作而输出的顶点属性
  float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
  vout.TexC = mul(texC, gMatTransform).xy;

  return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
  float4 diffuseAlbedo = gDiffuseMap.Sample(gsamAnisotropicWrap,
    pin.TexC) * gDiffuseAlbedo;

  // 对法线插值可能使之非规范化，因此要对它再次进行规范化处理
  pin.NormalW = normalize(pin.NormalW);

  // 光线经表面上一点反射到观察点这一方向上的向量
  float3 toEyeW = normalize(gEyePosW - pin.PosW);

  // 光照项
  float4 ambient = gAmbientLight*diffuseAlbedo;

  const float shininess = 1.0f - gRoughness;
  Material mat = { diffuseAlbedo, gFresnelR0, shininess };
  float3 shadowFactor = 1.0f;
  float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
    pin.NormalW, toEyeW, shadowFactor);

  float4 litColor = ambient + directLight;

  // 从漫反射反照率获取alpha值的常见手段
  litColor.a = diffuseAlbedo.a;

  return litColor;
}
```

9.10 纹理变换

我们还未曾讨论过常量缓冲区变量`gTexTransform`与`gMatTransform`。这两个变量用于在顶点着色器中对输入的纹理坐标进行变换：

```
// 为三角形插值而输出顶点属性
float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
vout.TexC = mul(texC, gMatTransform).xy;
```

纹理坐标表示的是纹理平面中的2D点。有了这种坐标，我们就能像其他的2D点一样，对纹理中的样点进行缩放、平移与旋转。下面是一些适用于纹理变换的示例：

1．令砖块纹理随着一堵墙的模型而拉伸。假设此墙顶点的当前纹理坐标范围为[0, 1]。在将该纹理坐标放大4倍使它们变换到范围[0, 4]时，该砖块纹理将沿着墙面重复贴图4×4次。

2．假设有许多云朵纹理绵延在万里无云的碧空背景之下，那么，通过随着时间函数来平移这些纹理的坐标，便能实现动态的白云浮过蔚蓝天空的效果。

3．纹理的旋转操作有时也便于实现一些类似于粒子的效果。例如，可以随着时间的推移而令火球纹理进行旋转。

在“Crate”（板条箱）演示程序中，我们实际上是采用单位矩阵进行变换，也就是说，并没有对输入的纹理坐标进行任何修改。但是，在下

一节里，我们会讲解一种使用纹理变换的示例。

注意，变换2D纹理坐标要利用4×4矩阵，因此我们先将它扩充为一个4D向量：

```
vin.TexC ---> float4(vin.TexC, 0.0f, 1.0f)
```

在完成乘法运算之后，从得到的4D向量中去掉z分量与w分量，使之强制转换回2D向量，即：

```
float4 TexC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
vout.TexC = mul(texC, gMatTransform).xy;
```

在此，我们运用了两个独立的纹理变换矩阵`gTexTransform`与`gMatTransform`，这样做是因为一种是关于材质的纹理变换（针对像水那样的动态材质），另一种是关于物体属性的纹理变换。

由于这里使用的是2D纹理坐标，所以我们只关心前两个坐标轴的变换情况。例如，如果纹理矩阵平移了z坐标，这并不会对纹理坐标造成任何影响。

9.11 附有纹理的山川演示程序

在此演示程序中，我们向陆地与河流的场景中添加了纹理。首先要解决的问题是向陆地铺设草地纹理。由于陆地的网格是一个块极大的曲面，若简单地沿着它的形状在上面拉伸纹理，将导致每个三角形仅分配到极少的纹素。换句话说，此做法并没有给予表面足够的纹理分辨率，所以最终只能得到纹理放大的失真效果。因此，这时就要通过向陆地网格重复铺设草地纹理来获取更高的分辨率。第二个关键问题则是根据时间函数令水流纹理沿波浪几何体滚（“流”）动起来。添加此项动作可使流水的效果更加逼真。图9.15为此演示程序的效果。

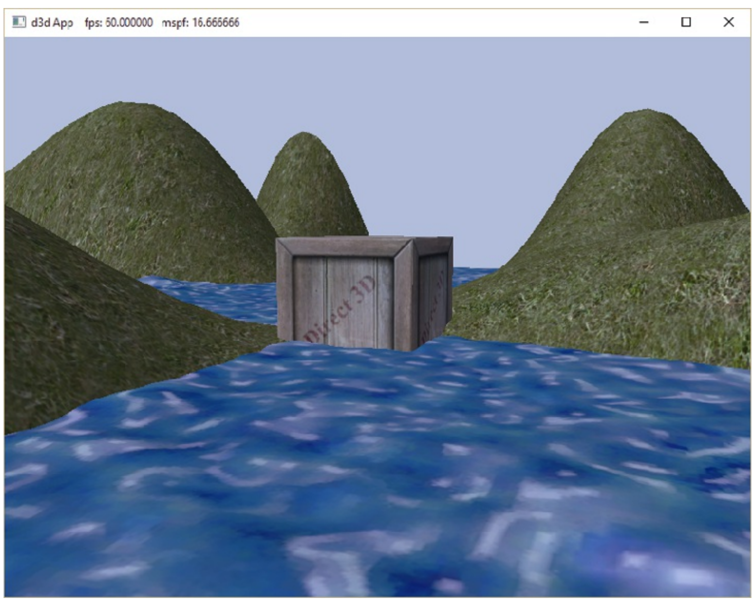


图9.15　山川纹理例程的效果

9.11.1 生成栅格纹理坐标

图9.16左侧所示的是一个位于平面xz内的$m \times n$栅格，右侧则是在归一化纹理坐标域$[0,1]^2$中与之对应的栅格。从图中可以明显地看出，xz平面内的栅格顶点纹理坐标与纹理坐标系中的栅格顶点坐标一一对应。纹理坐标系中第i行、第j列的顶点坐标为：

$$u_{ij} = j \cdot \Delta u$$
$$v_{ij} = i \cdot \Delta v$$

其中，$\Delta u = \dfrac{1}{n-1}$，$\Delta v = \dfrac{1}{m-1}$。

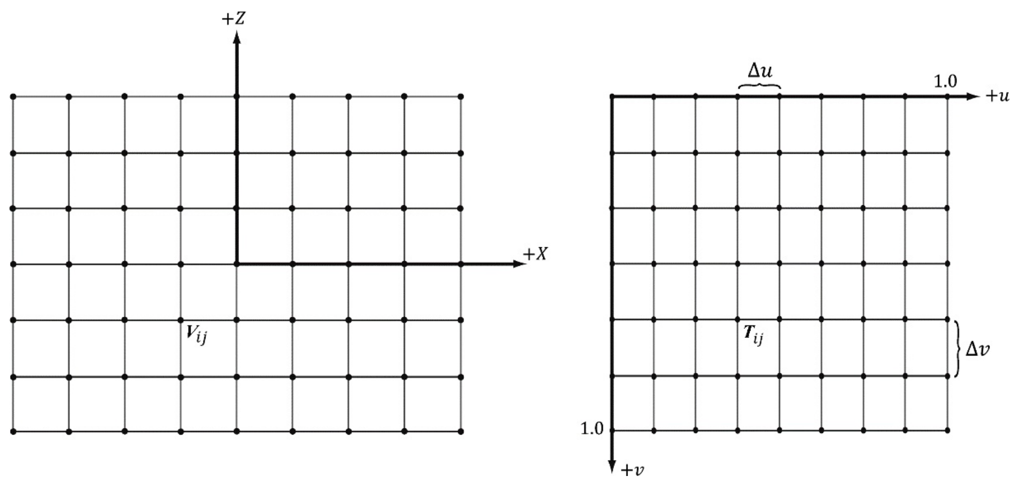


图9.16　xz空间内栅格顶点$\boldsymbol{v}_{ij}$的纹理坐标对应于uv纹理坐标系中栅格顶点$\boldsymbol{T}_{ij}$的坐标

据此，我们就可以通过下列`GeometryGenerator::CreateGrid`方法来生成栅格的纹理坐标。

```
GeometryGenerator::MeshData
GeometryGenerator::CreateGrid(float width, float depth, uint32 m, uint32 n
)
{
  MeshData meshData;

  uint32 vertexCount = m*n;
  uint32 faceCount  = (m-1)*(n-1)*2;
```

```
  float halfWidth = 0.5f*width;
  float halfDepth = 0.5f*depth;

  float dx = width / (n-1);
  float dz = depth / (m-1);

  float du = 1.0f / (n-1);
  float dv = 1.0f / (m-1);

  meshData.Vertices.resize(vertexCount);
  for(uint32 i = 0; i < m; ++i)
  {
    float z = halfDepth - i*dz;
    for(uint32 j = 0; j < n; ++j)
    {
      float x = -halfWidth + j*dx;

      meshData.Vertices[i*n+j].Position = XMFLOAT3(x, 0.0f, z);
      meshData.Vertices[i*n+j].Normal  = XMFLOAT3(0.0f, 1.0f, 0.0f);
      meshData.Vertices[i*n+j].TangentU = XMFLOAT3(1.0f, 0.0f, 0.0f);

      // 根据栅格拉伸纹理
      meshData.Vertices[i*n+j].TexC.x = j*du;
      meshData.Vertices[i*n+j].TexC.y = i*dv;
    }
  }
```

9.11.2 铺设纹理

正如之前所述，我们现要在陆地网格上铺设草丛纹理。但是到目前为止，我们所计算的纹理坐标仅限于单位域$[0, 1]^2$内，所以仅凭这一点还无法完成纹理的铺设。为此，我们要指定重复寻址模式，并通过纹理变换矩阵使纹理坐标按比例放大5倍。如此一来，纹理坐标就将被映射到区间$[0, 5]^2$之中，而纹理也就可以在陆地网格曲面上铺设5×5次：

```
void TexWavesApp::BuildRenderItems()
{
  auto gridRitem = std::make_unique<RenderItem>();
  gridRitem->World = MathHelper::Identity4x4();
```

```
  XMStoreFloat4x4(&gridRitem->TexTransform,
    XMMatrixScaling(5.0f, 5.0f, 1.0f));
  ...
}
```

9.11.3　纹理动画

为了使水流纹理（可以像滚动广告那样）顺着波浪几何体滚动播放，我们要在每个更新周期中调用AnimateMaterials方法，以此根据时间函数在纹理平面内平移纹理坐标。每帧中的位移量要小，这样可以使动画看上去更加平滑流畅。我们用无缝纹理按重复寻址模式进行贴图，这样就能沿着纹理坐标系平面接连不断并不露痕迹地平移纹理坐标。下列代码演示了我们是如何来计算水流纹理的偏移向量，且如何来构建并设置流水的纹理矩阵。

```
void TexWavesApp::AnimateMaterials(const GameTimer& gt)
{
   // 使水流材质的纹理坐标滚动起来
   auto waterMat = mMaterials["water"].get();

   float& tu = waterMat->MatTransform(3, 0);
   float& tv = waterMat->MatTransform(3, 1);

   tu += 0.1f * gt.DeltaTime();
   tv += 0.02f * gt.DeltaTime();

   if(tu >= 1.0f)
     tu -= 1.0f;

   if(tv >= 1.0f)
     tv -= 1.0f;

   waterMat->MatTransform(3, 0) = tu;
   waterMat->MatTransform(3, 1) = tv;

   // 材质已发生变化，因而需要更新常量缓冲区
   waterMat->NumFramesDirty = gNumFrameResources;
```

```
}
```

9.12　小结

1．纹理坐标用于定义将要映射到3D三角形上的纹理三角形。

2．对于游戏而言，创建纹理的常用方法是，请贴图师在Photoshop或一些其他的图像编辑器中进行创作，并将结果存为某种图像文件，如BMP、DDS、TGA或PNG。接着，游戏应用程序会在加载资源期间将图像数据载入`ID3D12Resource`对象。对于实时图形应用程序来说，DDS（DirectDraw图面格式）图像文件是极好的选择，因为它支持各种GPU本身即可处理的图像格式，以及GPU可原生解压的压缩图像格式。

3．有两种将传统图像格式转换为DDS格式的常用方法：使用图像编辑器或借助一种微软公司提供的名为texconv的命令行工具导出DDS格式。

4．我们能够通过`CreateDDSTextureFromFile12`函数以存于磁盘中的图像文件来创建纹理，此函数位于DVD中的Common/DDSTextureLoader.h/.cpp文件。

5．当放大物体表面并试图以少数纹素来覆盖大量屏幕像素时，便会涉及纹理放大（magnification）的问题。而当缩小物体表面并尝试令大量纹素覆盖少数屏幕像素时，就会进行纹理缩小（minification）的相关操作。mipmap与纹理过滤器是处理纹理放大与缩小这两种操作的关键技术。GPU原生支持3种纹理过滤器（根据质量由低到高、开销由廉至贵的顺序来排列），即点过滤器、线性过滤器以及各向异性过滤器。

6．纹理的寻址模式定义了Direct3D将如何处理超出范围[0, 1]外的纹理坐标。例如，对于那些超出范围的纹理，究竟是应当采取平铺、镜像，还是钳位或其他处理方式呢？

7．我们可以像变换普通点那样，利用纹理坐标对纹理进行缩放、旋转以及平移。通过在每一帧中小幅度渐进地变换纹理坐标，便可以实现纹理的动画效果。

9.13　练习

1．尝试修改“Crate”（板条箱）演示程序中的纹理坐标，并采用不同的寻址模式与过滤选项来开展试验。特别是要再现图9.7、图9.9～图9.13中的效果。

2．通过使用DirectX纹理工具（DirectX Texture Tool）[7]，我们就能自己来指定每个mipmap层级（**File→Open Onto This Surface**）。创建一个具有如图9.17中所示mipmap链的DDS文件，在每一层级中附有不同的文字说明或颜色能便于我们区分出每一个mipmap层。通过此纹理来代替Crate演示程序中的原纹理后，在镜头拉近或远离板条箱的过程中，我们便能明显地看出mipmap层级的变化。最后，分别尝试mipmap的点过滤与线性过滤，并观察效果。

3．给出两个相同大小的纹理，我们可以通过各种不同的处理方式将它们合成为一个新的图像。一般来讲，这种将多个纹理合而为一的方法称为多重纹理贴图（multitexturing）。例如，我们可以对两个纹理中相应的纹素进行加、减或（分量式）乘法运算。图9.18展示了通过将两种纹理进行分量式乘法来获取火球效果的过程。在本练习中要修改的是“Crate”演示程序，用像素着色器令图9.18内所示的两种原始纹理素材组合为火球纹理，并将其映射到每个立方体的表面上（此练习中的图像文件可以从本书的配套网站上下载[8]）。注意，我们需要修改*Default.hlsl*文件来支持处理多个纹理。



图9.17　手动构建的mipmap链，这样一来每个层级都能看得通透

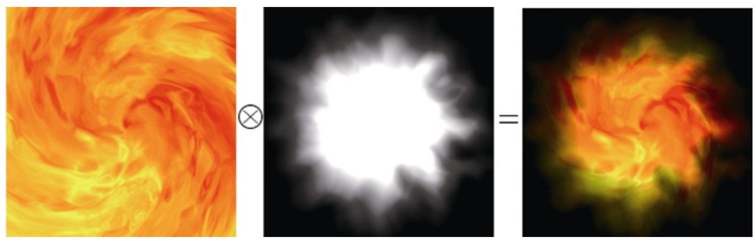


图9.18　通过使两个纹理中的对应纹素逐分量相乘来生成一个新的纹理

4．修改练习3的解答方案，使每个立方体表面上的火球纹理随着时间函数而旋转。

5．设点$\boldsymbol{p}_0$、$\boldsymbol{p}_1$与$\boldsymbol{p}_2$为三角形的顶点，而且它们分别对应于纹理坐标$\boldsymbol{q}_0$、$\boldsymbol{q}_1$和$\boldsymbol{q}_2$。回顾第9.2节可知，当$s \geqslant 0$，$t \geqslant 0$，$s+t \leqslant 1$时，对于3D三角形中任意一点$\boldsymbol{p}(s,t)=\boldsymbol{p}_0+s(\boldsymbol{p}_1-\boldsymbol{p}_0)+t(\boldsymbol{p}_2-\boldsymbol{p}_0)$，我们可用同样的参数$s$、$t$，通过对3D三角形顶点纹理坐标进行线性插值来求出其纹理坐标(u,v)：

$$(u,v)=\boldsymbol{q}_0+s(\boldsymbol{q}_1-\boldsymbol{q}_0)+t(\boldsymbol{q}_2-\boldsymbol{q}_0)$$

（a）给出(u,v)以及$\boldsymbol{q}_0$、$\boldsymbol{q}_1$和$\boldsymbol{q}_2$，用u与v来表示(s,t)。（提示：考虑向量方程$(u,v)=\boldsymbol{q}_0+s(\boldsymbol{q}_1-\boldsymbol{q}_0)+t(\boldsymbol{q}_2-\boldsymbol{q}_0)$。）

（b）将$\boldsymbol{p}$表示为由u与v作为自变量的函数，即求出公式$\boldsymbol{p}=\boldsymbol{p}(u,v)$

。

（c）计算$\partial \boldsymbol{P}/\partial u$与$\partial \boldsymbol{P}/\partial v$，并解释这些向量的几何意义。

6．修改第 8 章中的“LitColumns”演示程序，向场景中的地面、立柱以及球体添加纹理（营造图9.19中的效果）。这些纹理可在本章的代码目录中找到。[9]

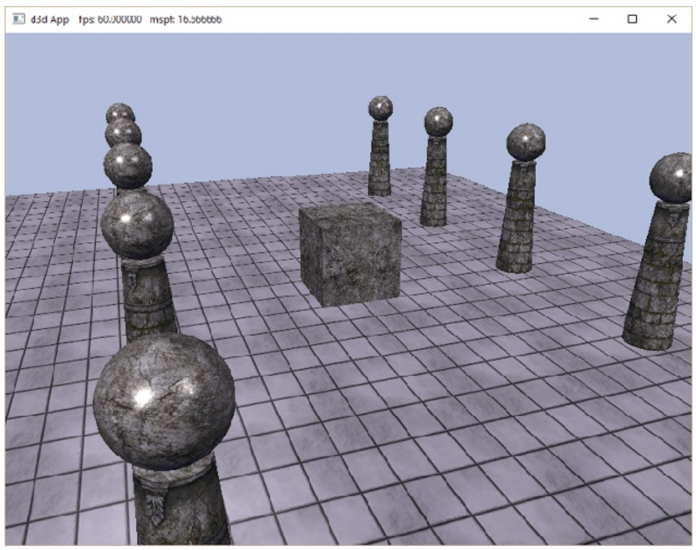


图9.19　对场景中的物体进行贴图后的效果

---

[1]　排除一些特殊的情况，这里所说的“矩阵”可看作是存有纹理数据元素的2D数组。

[2]　纹理空间这个提法不太好，容易与第19.3节中的切空间（亦称纹理空间）混淆。

[3]　最新版的texconv与texassemble工具可从微软GitHub上名为DirectXTex的工程中找到。

[4] 在翻译这段话的时候，已有支持DirectX 12的相关代码了。读者可在微软的GitHub中查找DirectXTK12，其中有对应版本的DDSTextureLoader.h/cpp文件。

[5] 过滤器，即filter。由于纹理过滤以及后面会提到的高斯模糊等（其实还包括反锯齿等许多与图形相关的技术）在本质上都运用的是信号处理技术，因此有时将其译作“滤波器”。考虑到从效果上来划分的话，有时也称高斯模糊等为“滤镜”。

[6] 有些文献将mipmap写作mip-map或直接作mip map，这样一看比较直观。可见它也是一种“图”，但是这是一种由一组内容相同大小不同的图所构成的“图”。

[7] DirectX Texture Tool的程序名为DxTex.exe，是DirectX 10之前的产物，官方网站已不再建议使用此工具。其实此程序随DirectX的更替也有许多版本，每个版本都有或多或少的bug，但是最严重问题的还是不支持DirectX 9后新出的图像格式。其优点便是可视化且易于操作。它是老版DirectX SDK中的一部分，所以要使用就需装整个SDK。网上独立提取出来的版本都比较低，如果读者当真要用的话，小的推荐使用GitHub中SourceEngine2007工程的DxTex.exe版本。原作者在此项目中附有许多有用的工具，版本也比较新，而且源码也提取出来了，可以自己动手丰衣足食，修改bug、添加图像格式等。而DXSDK_Jun10.exe是最后一版独立的DirectX SDK，从Windows 8开始DirectX SDK便归于Windows SDK。到了Visual Studio 2015这个版本，它已内置能够简单编辑图像的程序，可以将图片放进去试一试。在DirectX 12这个版本中，

DDS文件的处理则交由DirectXTex库，再辅以其他轻量级程序配合使用。这些常用的程序有文中所述的用于转换图像格式的Texconv，创建立方体图、体贴图以及纹理数组的Texassemble，还有查看DDS图像的DDSView等。简单来说，此版本以DirectXTex库为核心的命令行工具集代替了旧版本的可视化DxTex。另外，本书所推荐的NVIDIA版Photoshop插件也有格式支持不完整的问题，而微软官方GitHub则提到另一款由Intel制作的名为Texture Works Plugin的Photoshop插件，而且它也基于DirectXTex库。

[8]　这两种纹理可在本书前一版的源代码压缩包d3d11CodeSet1.zip的第8章中找到，名称分别为flare.dds和flarealpha.dds。

[9]　本书的纹理都统一放在Textures文件夹中。若所需纹理不在其中，可以如之前所说在此书前一版源代码的相应章节中寻找。

第10章　混合

先来观察一下图10.1。我们在渲染此帧画面时先后绘制了地形与木板箱（构成箱子在前、地形在后的效果），使这两种材质的像素数据都位于后台缓冲区中。接着再运用混合技术将水面绘制到后台缓冲区，令水的像素数据与地形以及板条箱这两种像素数据在后台缓冲区内相混合，构成可以透过水看到地形与板条箱的效果。本章将研究混合（blending，也译作融合）技术，它使我们可以将当前要光栅化（又名为源像素，source pixel）的像素与之前已光栅化至后台缓冲区的像素（目标像素，destination pixel）相融合。因此，该技术可用于渲染如水与玻璃之类的半透明物体。

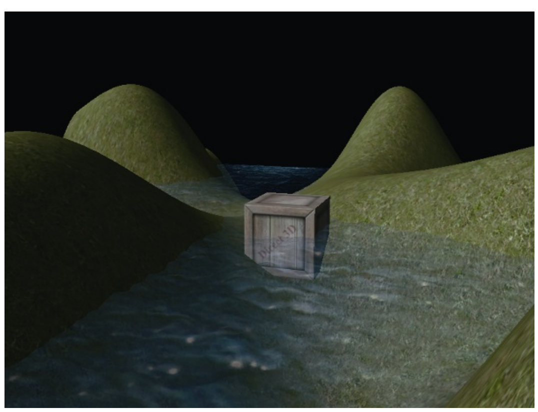


图10.1　半透明效果的水面

注意

为了便于讨论，我们将此处谈及的后台缓冲区视为渲染目标，而在本书的后面还会展示将物体渲染至“离屏”（off screen）的渲染目标之中。在这两种渲染目标中所运用混合技术其实并没有什么区别，而位于离屏渲染目标中的目标像素也是经此前光栅化处理后的像素数据而已。

学习目标：

1．理解混合技术的工作原理，并且在Direct3D中运用此技术。

2．学习Direct3D所支持的不同混合模式。

3．探究如何用alpha分量来调节图元的透明度。

4．学会仅通过调用HLSL中的`clip`函数来阻止向后台缓冲区中绘制像素。

10.1　混合方程

设$\mathbf{C}_{\mathrm{src}}$为像素着色器输出的当前正在光栅化的第$i$行、第$j$列像素（源像素）的颜色值，再设$\mathbf{C}_{\mathrm{dst}}$为目前在后台缓冲区中与之对应的第$i$行、第$j$列像素（目标像素）的颜色值。若不用混合技术，$\mathbf{C}_{\mathrm{src}}$将直接覆写$\mathbf{C}_{\mathrm{dst}}$（假设此像素已经通过深度/模板测试），而令后台缓冲区中第i行、第j列的像素变更为新的颜色值$\mathbf{C}_{\mathrm{src}}$。但是，若使用了混合技术，则$\mathbf{C}_{\mathrm{src}}$与$\mathbf{C}_{\mathrm{dst}}$将融合在一起得到新颜色值$\mathbf{C}$后再覆写$\mathbf{C}_{\mathrm{dst}}$（即将两者的混合颜色$\mathbf{C}$写入后台缓冲区的第$i$行、第$j$列像素）。Direct3D使用下列混合方程来使源像素颜色与目标像素颜色相融合：

$$\mathbf{C} = \mathbf{C}_{\mathrm{src}} \otimes \mathbf{F}_{\mathrm{src}} \boxplus \mathbf{C}_{\mathrm{dst}} \otimes \mathbf{F}_{\mathrm{dst}}$$

在第10.3节中将介绍$\mathbf{F}_{\mathrm{src}}$（源混合因子）与$\mathbf{F}_{\mathrm{dst}}$（目标混合因子）会使用到的具体值，通过这两种因子，我们就能够用各种数值来调整源像素与目标像素，以获取各种不同的效果。运算符“$\otimes$”表示在5.3.1节中针对颜色向量而定义的分量式乘法，而“$\boxplus$”则表示在10.2节中定义的二元运算符。

上述混合方程仅用于控制颜色的RGB分量，而alpha分量实则由类似于下面的方程来单独处理：

$$A = A_{\mathrm{src}} F_{\mathrm{src}} \boxplus A_{\mathrm{dst}} F_{\mathrm{dst}}$$

这两组方程本质上都是相同的，但区别在于混合因子与二元运算可

能有所差异。将RGB分量与alpha分量分离开来的动机也比较简单，就是希望能独立地处理两者，来尽可能多地产生不同的混合变化效果。

注意 Note

alpha分量的混合需求远少于RGB分量的混合需求。这主要是由于我们往往并不关心后台缓冲区中的alpha值。而仅在一些对目标alpha值（destination alpha）有特定要求的算法之中，后台缓冲区内的alpha值才显得至关重要。

10.2　混合运算

下列枚举项成员将用作混合方程中的二元运算符⊞：

```
typedef enum D3D12_BLEND_OP
{
  D3D12_BLEND_OP_ADD = 1,          $\boldsymbol{C} = \boldsymbol{C}_{\text{src}} \otimes \boldsymbol{F}_{\text{src}} \boldsymbol{C}_{\text{dst}} \otimes \boldsymbol{F}_{\text{dst}}$
  D3D12_BLEND_OP_SUBTRACT = 2,     $\boldsymbol{C} = \boldsymbol{C}_{\text{dst}} \otimes \boldsymbol{F}_{\text{dst}} - \boldsymbol{C}_{\text{src}} \otimes \boldsymbol{F}_{\text{src}}$
  D3D12_BLEND_OP_REV_SUBTRACT = 3, $\boldsymbol{C} = \boldsymbol{C}_{\text{src}} \otimes \boldsymbol{F}_{\text{src}} - \boldsymbol{C}_{\text{dst}} \otimes \boldsymbol{F}_{\text{dst}}$
  D3D12_BLEND_OP_MIN = 4,          $\boldsymbol{C} = \min(\boldsymbol{C}_{\text{src}}, \boldsymbol{C}_{\text{dst}})$
  D3D12_BLEND_OP_MAX = 5,          $\boldsymbol{C} = \max(\boldsymbol{C}_{\text{src}}, \boldsymbol{C}_{\text{dst}})$
  } D3D12_BLEND_OP;
```

注意　Note

在求取最小值或最大值（min/max）的运算中会忽略混合因子。

这些运算符也同样适用于alpha混合运算。而且，我们还能同时为RGB和alpha这两种运算分别指定不同的运算符。例如，可以像下面一样使两个RGB项相加，却令两个alpha项相减：

$$\begin{aligned} \boldsymbol{C} &= \boldsymbol{C}_{\text{src}} \otimes \boldsymbol{F}_{\text{src}} + \boldsymbol{C}_{\text{dst}} \otimes \boldsymbol{F}_{\text{dst}} \\ A &= A_{\text{dst}} F_{\text{dst}} - A_{\text{src}} A_{\text{src}} \end{aligned}$$

Direct3D从最近几版开始加入了一项新特性，通过逻辑运算符对源颜色和目标颜色进行混合，用以取代上述传统的混合方程。这些逻辑运

算符如下：

```
typedef
enum D3D12_LOGIC_OP
{
  D3D12_LOGIC_OP_CLEAR  = 0,
  D3D12_LOGIC_OP_SET    = ( D3D12_LOGIC_OP_CLEAR + 1 ) ,
  D3D12_LOGIC_OP_COPY   = ( D3D12_LOGIC_OP_SET + 1 ) ,
  D3D12_LOGIC_OP_COPY_INVERTED         = ( D3D12_LOGIC_OP_COPY + 1 ) ,
  D3D12_LOGIC_OP_NOOP   = ( D3D12_LOGIC_OP_COPY_INVERTED + 1 ) ,
  D3D12_LOGIC_OP_INVERT = ( D3D12_LOGIC_OP_NOOP + 1 ) ,
  D3D12_LOGIC_OP_AND    = ( D3D12_LOGIC_OP_INVERT + 1 ) ,
  D3D12_LOGIC_OP_NAND   = ( D3D12_LOGIC_OP_AND + 1 ) ,
  D3D12_LOGIC_OP_OR     = ( D3D12_LOGIC_OP_NAND + 1 ) ,
  D3D12_LOGIC_OP_NOR    = ( D3D12_LOGIC_OP_OR + 1 ) ,
  D3D12_LOGIC_OP_XOR    = ( D3D12_LOGIC_OP_NOR + 1 ) ,
  D3D12_LOGIC_OP_EQUIV  = ( D3D12_LOGIC_OP_XOR + 1 ) ,
  D3D12_LOGIC_OP_AND_REVERSE  = ( D3D12_LOGIC_OP_EQUIV + 1 ) ,
  D3D12_LOGIC_OP_AND_INVERTED = ( D3D12_LOGIC_OP_AND_REVERSE + 1 ) ,
  D3D12_LOGIC_OP_OR_REVERSE   = ( D3D12_LOGIC_OP_AND_INVERTED + 1 ) ,
  D3D12_LOGIC_OP_OR_INVERTED  = ( D3D12_LOGIC_OP_OR_REVERSE + 1 )
} D3D12_LOGIC_OP;
```

注意，不能同时使用传统混合方程与逻辑运算符这两种混合手段，两者只能择其一。另外需要指出的是，为了使用逻辑运算符混合技术，就一定要选择它所支持的渲染目标格式——这个格式应当为UINT（无符号整数）的有关类型，否则我们会收到类似于下面的错误提示信息：

```
D3D12 ERROR: ID3D12Device::CreateGraphicsPipelineState: The render
target format at slot 0 is format (R8G8B8A8_UNORM). This format
does not support logic ops. The Pixel Shader output signature
indicates this output could be written, and the Blend State indicates
logic op is enabled for this slot. [ STATE_CREATION ERROR #678:
CREATEGRAPHICSPIPELINESTATE_OM_RENDER_TARGET_DOES_NOT_SUPPORT_LOGIC_OPS
 ]

D3D12 WARNING: ID3D12Device::CreateGraphicsPipelineState: Pixel Shader
output 'SV_Target0' has type that is NOT unsigned int, while the
corresponding Output Merger RenderTarget slot [0] has logic op enabled.
This happens to be well defined: the raw bits output from the shader
will simply be interpreted as UINT bits in the blender without any data
```

conversion. This warning is to check that the application developer really intended to rely on this behavior. [STATE_CREATION WARNING #677: CREATEGRAPHICSPIPELINESTATE_PS_OUTPUT_TYPE_MISMATCH]

（D3D12错误： ID3D12Device::CreateGraphicsPipelineState： 位于0槽位的渲染目标格式为（R8G8B8A8_UNORM）。逻辑运算不支持此格式。像素着色器的输出签名指示可以向此输出项执行写操作，而且混合状态表明此槽位启用的是逻辑运算。[STATE_CREATION ERROR #678:CREATEGRAPHICSPIPELINESTATE_ OM_RENDER_TARGET_DOES_NOT_SUPPORT_LOGIC_OPS]

D3D12警告：ID3D12Device::CreateGraphicsPipelineState：像素着色器输出项'SV_Target0'存在非无符号整数类型，而它相应的输出合并渲染目标槽[0]却已经启用了逻辑运算。关于此事件的发生系统中有着明确的定义：从着色器输出的原始二进制数据在混合器中将不进行任何的数据转换，而是简单地解释为UINT二进制数据。此警告是为了核实应用程序开发者是否遵循此行为。[STATE_CREATION WARNING #677:CREATEGRAPHICSPIPELINESTATE_ PS_OUTPUT_TYPE_MISMATCH]）

10.3 混合因子

通过为源混合因子与目标混合因子分别设置不同的混合运算符，就可以实现各式各样的混合效果。我们会在10.5节中详解这些组合，但是需要先来体验一下不同的混合因子并感受它们实际的计算方式。下面列举描述的是基本的混合因子，可以将它们应用于$\boldsymbol{F}_{\mathrm{src}}$与$\boldsymbol{F}_{\mathrm{dst}}$。关于其他更加高级的混合因子，读者可参考SDK文档中的D3D12_BLEND枚举类型。设$\boldsymbol{C}_{\mathrm{src}}=(r_{\mathrm{s}},g_{\mathrm{s}},b_{\mathrm{s}})A_{\mathrm{src}}=a_{\mathrm{s}}$（从像素着色器输出的RGBA值），$\boldsymbol{C}_{\mathrm{dst}}=(r_{\mathrm{d}},g_{\mathrm{d}},b_{\mathrm{d}})A_{\mathrm{dst}}=a_{\mathrm{d}}$（已存储于渲染目标中的RGBA值），$\boldsymbol{F}$为$\boldsymbol{F}_{\mathrm{src}}$或$\boldsymbol{F}_{\mathrm{dst}}$，而$F$是$F_{\mathrm{src}}$或$F_{\mathrm{dst}}$，则我们有：

D3D12_BLEND_ZERO: $\boldsymbol{F}=(0,0,0)$且$F=0$

D3D12_BLEND_ONE: $\boldsymbol{F}=(1,1,1)$且$F=1$

D3D12_BLEND_SRC_COLOR: $\boldsymbol{F}=(r_{\mathrm{s}},g_{\mathrm{s}},b_{\mathrm{s}})$

D3D12_BLEND_INV_SRC_COLOR: $\boldsymbol{F}_{\mathrm{src}}=(1-r_{\mathrm{s}},1-g_{\mathrm{s}},1-b_{\mathrm{s}})$

D3D12_BLEND_SRC_ALPHA:$\boldsymbol{F}=(a_{\mathrm{s}},a_{\mathrm{s}},a_{\mathrm{s}})$且$F=a_{\mathrm{s}}$

D3D12_BLEND_INV_SRC_ALPHA: $\boldsymbol{F}=(1-a_{\mathrm{s}},1-a_{\mathrm{s}},1-a_{\mathrm{s}})$且$F=(1-a_{\mathrm{s}})$

D3D12_BLEND_DEST_ALPHA: $\boldsymbol{F}=(a_{\mathrm{d}},a_{\mathrm{d}},a_{\mathrm{d}})$且$F=a_{\mathrm{d}}$

D3D12_BLEND_INV_DEST_ALPHA: $\boldsymbol{F}=(1-a_{\mathrm{d}},1-a_{\mathrm{d}},1-a_{\mathrm{d}})$且

$F = (1 - a_d)$

D3D12_BLEND_DEST_COLOR: $\boldsymbol{F} = (r_d, g_d, b_d)$

D3D12_BLEND_INV_DEST_COLOR: $\boldsymbol{F} = (1 - r_d, 1 - g_d, 1 - b_d)$

D3D12_BLEND_SRC_ALPHA_SAT: $\boldsymbol{F} = (a'_s, a'_s, a'_s)$且$F = a'_s$，其中$a'_s = \text{clamp}(a_s,\ 0,\ 1)$

D3D12_BLEND_BLEND_FACTOR：$\boldsymbol{F} = (r, g, b)$且$F = a$，其中的颜色$(r, g, b, a)$可用作方法ID3D12GraphicsCommandList::OMSetBlendFactor的参数。通过这种方法，我们就可以直接指定所用的混合因子值。但是在改变混合状态（blend state）之前，此值是不会生效的。

D3D12_BLEND_INV_BLEND_FACTOR：$\boldsymbol{F} = (1 - r, 1 - g, 1 - b)$且$F = 1 - a$，这里的颜色（$r, g, b, a$）可用作ID3D12GraphicsCommandList::OMSetBlendFactor的参数，这使我们可以直接指定所用的混合因子值。然而，在混合状态变化之前，此值保持不变。

上述的混合因子皆可运用于RGB混合方程。但对于alpha混合方程来说，却不可使用以_COLOR作为结尾的混合因子。

注意 Note

`clamp`函数的定义为：

$$\text{clamp}(x, a, b) = \begin{cases} x, & a \le x \le b \\ a, & x < a \\ b, & x > b \end{cases}$$

注意

我们可以用下列函数来设置混合因子：

```
void ID3D12GraphicsCommandList::OMSetBlendFactor(
  const FLOAT BlendFactor[ 4 ]);
```

若传入`nullptr`，则恢复值为(1, 1, 1, 1)的默认混合因子。

10.4　混合状态

前面已经讨论过混合运算符与混合因子，那么怎样用Direct3D来设置这些数值呢？就像其他的Direct3D状态一样，混合状态亦是PSO（流水线状态对象）的一部分。到目前为止，我们一直使用的都是默认的混合状态，即并没有启用混合技术：

```
D3D12_GRAPHICS_PIPELINE_STATE_DESC opaquePsoDesc;
ZeroMemory(&opaquePsoDesc, sizeof(D3D12_GRAPHICS_PIPELINE_STATE_DESC));
...
opaquePsoDesc.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT);
```

为了配置非默认混合状态，我们必须填写D3D12_BLEND_DESC结构体。该结构体的定义如下。

```
typedef struct D3D12_BLEND_DESC {
 BOOL AlphaToCoverageEnable;   // 默认值为 False
 BOOL IndependentBlendEnable;  // 默认值为 False
 D3D12_RENDER_TARGET_BLEND_DESC RenderTarget[8];
} D3D12_BLEND_DESC;
```

1．AlphaToCoverageEnable：指定为true，则启用alpha-to-coverage功能，这是一种在渲染叶片或门等纹理时极其有用的一种多重采样技术。若指定为false，则禁用alpha-to-coverage功能。另外，要使用此技术还需开启多重采样（即创建后台缓冲区与深度缓冲区时要启用多重采样）。

2．IndependentBlendEnable：Direct3D最多可同时支持8个渲染目标。若此标志被置为true，即表明可以向每一个渲染目标执行不同的

混合操作（不同的混合因子、不同的混合运算以及设置不同的混合禁用或开启状态等）。如果将此标志设为false，则意味着所有的渲染目标均使用D3D12_BLEND_DESC::RenderTarget数组中第一个元素所描述的方式进行混合。多渲染目标技术常用于高级算法，而现在我们只假设每次仅向一个渲染目标进行绘制。

3．RenderTarget：具有8个D3D12_RENDER_TARGET_BLEND_DESC元素的数组，其中的第i个元素描述了如何针对第i个渲染目标进行混合处理。如果IndependentBlendEnable被设置为false，则所有的渲染目标都将根据RenderTarget[0]的设置进行混合运算。

结构体D3D12_RENDER_TARGET_BLEND_DESC的定义如下。

```
typedef struct D3D12_RENDER_TARGET_BLEND_DESC
{
  BOOL BlendEnable; // 默认值为 False
  BOOL LogicOpEnable; // 默认值为 False
  D3D12_BLEND SrcBlend; // 默认值为 D3D12_BLEND_ONE
  D3D12_BLEND DestBlend; // 默认值为 D3D12_BLEND_ZERO
  D3D12_BLEND_OP BlendOp; // 默认值为 D3D12_BLEND_OP_ADD
  D3D12_BLEND SrcBlendAlpha; // 默认值为 D3D12_BLEND_ONE
  D3D12_BLEND DestBlendAlpha; // 默认值为 D3D12_BLEND_ZERO
  D3D12_BLEND_OP BlendOpAlpha; // 默认值为 D3D12_BLEND_OP_ADD
  D3D12_LOGIC_OP LogicOp;   // 默认值为 D3D12_LOGIC_OP_NOOP
  UINT8 RenderTargetWriteMask; // 默认值为 D3D12_COLOR_WRITE_ENABLE_ALL
} D3D12_RENDER_TARGET_BLEND_DESC;
```

1．BlendEnable：指定为true，则启用常规混合功能；指定为false，则禁用常规混合功能。注意，不能将BlendEnable与LogicOpEnable同时置为true，只能从常规混合与逻辑运算符混合两种方式中选择一种。

2. LogicOpEnable：指定为true，则启用逻辑混合运算，反之则反。注意，不能将BlendEnable和LogicOpEnable同时设置为true，只能从常规混合与逻辑运算混合中选择一种。

3. SrcBlend：枚举类型D3D12_BLEND中的成员之一，用于指定RGB混合中的源混合因子$\boldsymbol{F}_{\mathrm{src}}$。

4. DestBlend：枚举类型D3D12_BLEND中的成员之一，用于指定RGB混合中的目标混合因子$\boldsymbol{F}_{\mathrm{dst}}$。

5. BlendOp：枚举类型D3D12_BLEND_OP中的成员之一，用于指定RGB混合运算符。

6. SrcBlendAlpha：枚举类型D3D12_BLEND中的一个成员，指定了alpha混合中的源混合因子F_{src}。

7. DestBlendAlpha：枚举类型D3D12_BLEND中的一个成员，指定了alpha混合中的目标混合因子F_{dst}。

8. BlendOpAlpha：枚举类型D3D12_BLEND_OP中的一个成员，指定了alpha混合运算符。

9. LogicOp：枚举类型D3D12_LOGIC_OP中的成员之一，指定了源颜色与目标颜色在混合时所用的逻辑运算符。

10. RenderTargetWriteMask：下列标志中一种或多种的组合。

```
typedef enum D3D12_COLOR_WRITE_ENABLE {
```

```
 D3D12_COLOR_WRITE_ENABLE_RED    = 1,
 D3D12_COLOR_WRITE_ENABLE_GREEN  = 2,
 D3D12_COLOR_WRITE_ENABLE_BLUE   = 4,
 D3D12_COLOR_WRITE_ENABLE_ALPHA  = 8,
 D3D12_COLOR_WRITE_ENABLE_ALL    =
   ( D3D12_COLOR_WRITE_ENABLE_RED | D3D12_COLOR_WRITE_ENABLE_GREEN |
    D3D12_COLOR_WRITE_ENABLE_BLUE | D3D12_COLOR_WRITE_ENABLE_ALPHA )
} D3D12_COLOR_WRITE_ENABLE;
```

这些标志控制着混合后的数据可被写入后台缓冲区中的哪些颜色通道。例如，通过指定D3D12_COLOR_WRITE_ENABLE_ALPHA可以禁止向RGB通道的写操作，而仅写入alpha通道的有关数据。对于一些高级技术而言，这种灵活性是极其实用的。当混合功能被禁止时，从像素着色器返回的颜色数据将按没有设置上述写掩码来进行处理（即不对目标像素执行任何操作）。

注意 Note

混合运算并非没有开销，它也需要对每个像素进行额外的处理。所以只有在需要的情况下才使用此技术，否则应禁用这项功能。

下面的代码展示了如何来创建和设置混合状态。

```
// 创建开启混合功能的PSO
D3D12_GRAPHICS_PIPELINE_STATE_DESC transparentPsoDesc = opaquePsoDesc;

D3D12_RENDER_TARGET_BLEND_DESC transparencyBlendDesc;
transparencyBlendDesc.BlendEnable = true;
transparencyBlendDesc.LogicOpEnable = false;
transparencyBlendDesc.SrcBlend = D3D12_BLEND_SRC_ALPHA;
```

```
transparencyBlendDesc.DestBlend = D3D12_BLEND_INV_SRC_ALPHA;
transparencyBlendDesc.BlendOp = D3D12_BLEND_OP_ADD;
transparencyBlendDesc.SrcBlendAlpha = D3D12_BLEND_ONE;
transparencyBlendDesc.DestBlendAlpha = D3D12_BLEND_ZERO;
transparencyBlendDesc.BlendOpAlpha = D3D12_BLEND_OP_ADD;
transparencyBlendDesc.LogicOp = D3D12_LOGIC_OP_NOOP;
transparencyBlendDesc.RenderTargetWriteMask = D3D12_COLOR_WRITE_ENABLE_ALL
;

transparentPsoDesc.BlendState.RenderTarget[0] = transparencyBlendDesc;
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(
  &transparentPsoDesc, IID_PPV_ARGS(&mPSOs["transparent"])));
```

如同其他的PSO一般，我们应当在应用程序的初始化期间来创建它们，接着再根据需求以`ID3D12GraphicsCommandList::SetPipelineState`方法在不同的状态之间进行切换。

10.5　混合示例

在下面的各小节中，我们将考查一些用于获取特效的混合因子组合。在这些示例中，我们只关注RGB混合，而alpha混合的处理方法则与之相似。

10.5.1　禁止颜色的写操作

如果希望使原始的目标像素保持不变，既不对它进行覆写，也不与当前光栅化的源像素执行混合，那么这个示例会非常适用。比如说，若不涉及后台缓冲区，而只对深度/模板缓冲区进行写操作时，就把源像素的混合因子设置为D3D12_BLEND_ZERO，将目标混合因子配置为D3D12_BLEND_ONE，再令混合运算符为D3D12_BLEND_OP_ADD即可。根据这一系列设定，混合方程可化简为：

$$\boldsymbol{C} = \boldsymbol{C}_{\mathrm{src}} \otimes \boldsymbol{F}_{\mathrm{src}} + \boldsymbol{C}_{\mathrm{dst}} \otimes \boldsymbol{F}_{\mathrm{dst}}$$

$$\boldsymbol{C} = \boldsymbol{C}_{\mathrm{src}} \otimes (0,0,0) + \boldsymbol{C}_{\mathrm{dst}} \otimes (1,1,1)$$

$$\boldsymbol{C} = \boldsymbol{C}_{\mathrm{dst}}$$

这是一个为便于讲解而精心设计的示例。其实还有一种能实现相同功能的简便方法，即将成员D3D12_RENDER_TARGET_BLEND_DESC::RenderTargetWriteMask设置为0，以此来禁止向任何颜色通道执行的写操作。

10.5.2　加法混合与减法混合

如果希望令源像素与目标像素实现加法运算（见图10.2），那么就将源混合因子与目标混合因子同设为D3D12_BLEND_ONE，再把混合运算符置为D3D12_BLEND_OP_ADD。对于这种配置而言，混合方程可化简为：

$$\boldsymbol{C} = \boldsymbol{C}_{\text{src}} \otimes \boldsymbol{F}_{\text{src}} \boxplus \boldsymbol{C}_{\text{dst}} \otimes \boldsymbol{F}_{\text{dst}}$$

$$\boldsymbol{C} = \boldsymbol{C}_{\text{src}} \otimes (1,1,1) + \boldsymbol{C}_{\text{dst}} \otimes (1,1,1)$$

$$\boldsymbol{C} = \boldsymbol{C}_{\text{src}} + \boldsymbol{C}_{\text{dst}}$$

另外，还可以继续使用上述混合因子，唯令D3D12_BLEND_OP_SUBTRACT来取代其中的加法混合运算符，以此来达到从目标像素中减去源像素这一目的（见图10.3）。

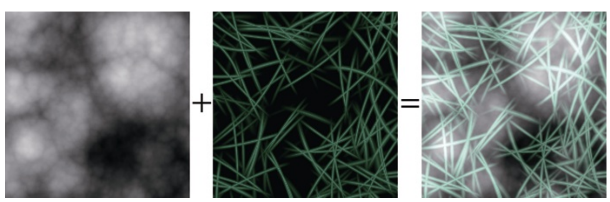


图10.2　令源颜色与目标颜色相加。由于加法运算总体提升了颜色值，因而生成了一张更为明亮的图像

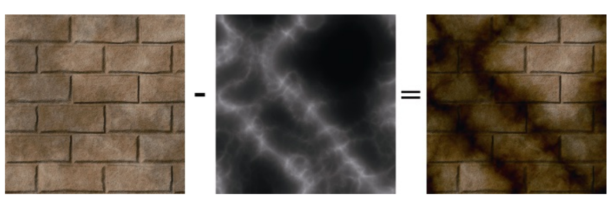


图10.3　从目标颜色中减去源颜色。由于减法运算移除了部分颜色信息，因而得到的图像更暗

10.5.3　乘法混合

如果希望将源像素与其对应的目标像素相乘（见图10.4），那么应设源混合因子为D3D12_BLEND_ZERO、目标混合因子为D3D12_BLEND_SRC_COLOR，再将混合运算符置为D3D12_BLEND_OP_ADD。据此配置，混合方程可化简为：

$$\boldsymbol{C} = \boldsymbol{C}_{\mathrm{src}} \otimes \boldsymbol{F}_{\mathrm{src}} \boxplus \boldsymbol{C}_{\mathrm{dst}} \otimes \boldsymbol{F}_{\mathrm{dst}}$$

$$\boldsymbol{C} = \boldsymbol{C}_{\mathrm{src}} \otimes (0,0,0) + \boldsymbol{C}_{\mathrm{dst}} \otimes \boldsymbol{C}_{\mathrm{src}}$$

$$\boldsymbol{C} = \boldsymbol{C}_{\mathrm{dst}} \otimes \boldsymbol{C}_{\mathrm{src}}$$



图10.4　令源颜色与目标颜色相乘

10.5.4　透明混合

设源alpha分量a_s为一种可用来控制源像素不透明度的百分比（例如，alpha为0表示100%透明，0.4表示40%不透明，1.0则表示100%不透明）。不透明度（opacity）与透明度（transparency）的关系很简单，即$T = 1 - A$，其中的A为不透明度，T为透明度。例如，若物体的不透明度达0.4，则透明度为1 − 0.4 = 0.6。现在，假设我们希望基于源像素的不透明度，将源像素与目标像素进行混合。为了实现此效果，设源混合因子为D3D12_BLEND_SRC_ALPHA、目标混合因子

为D3D12_BLEND_INV_SRC_ALPHA，并将混合运算符置为D3D12_BLEND_OP_ADD。有了这些配置，混合方程便可化简为：

$$\boldsymbol{C} = \boldsymbol{C}_{\mathrm{src}} \otimes \boldsymbol{F}_{\mathrm{src}} \boxplus \boldsymbol{C}_{\mathrm{dst}} \otimes \boldsymbol{F}_{\mathrm{dst}}$$

$$\boldsymbol{C} = \boldsymbol{C}_{\mathrm{src}} \otimes (a_{\mathrm{s}}, a_{\mathrm{s}}, a_{\mathrm{s}}) + \boldsymbol{C}_{\mathrm{dst}} \otimes (1 - a_{\mathrm{s}}, 1 - a_{\mathrm{s}}, 1 - a_{\mathrm{s}})$$

$$\boldsymbol{C} = a_{\mathrm{s}} \boldsymbol{C}_{\mathrm{src}} + (1 - a_{\mathrm{s}}) \boldsymbol{C}_{\mathrm{dst}}$$

例如，假设$a_s = 0.25$，也就是说，源像素的不透明度仅为25%。由于源像素的透明度为75%，因此，这就是说，在源像素与目标像素混合在一起的时候，我们便希望最终颜色将由25%的源像素与75%的目标像素组合而成（目标像素会位于源像素的“后侧”）。根据上述方程精确地推导出下列混合计算过程：

$$\boldsymbol{C} = a_s \boldsymbol{C}_{src} + (1 - a_s) \boldsymbol{C}_{dst}$$

$$\boldsymbol{C} = 0.25 \boldsymbol{C}_{src} + 0.75 \boldsymbol{C}_{dst}$$

借助此混合方法，我们就能绘制出类似于图10.1中那样的透明物体。需要注意的是，在使用此混合方法时，还应当考虑物体的绘制顺序。对此，我们应遵循以下规则：

首先要绘制无需混合处理的物体。接下来，再根据混合物体与摄像机的距离对它们进行排序。最后，按由远及近的顺序通过混合的方式来绘制这些物体。

依照由后向前的顺序进行绘制的原因是，每个物体都会与其后的所有物体执行混合运算。对于一个透明的物体而言，我们应当可以透过它看到其背后的场景。因此就需要将透明物体后的所有对应像素都预先写

入后台缓冲区内，随后再将此透明物体的源像素与其后场景的目标像素进行混合。

而对于10.5.1节中所述的渲染方法而言，绘制顺序便是无关紧要的了，因为它可以轻而易举地阻止源像素写入后台缓冲区。但针对10.5.2节与10.5.3节中所讨论的混合方法来说，我们照理就要先绘制非混合物体，最后再绘制需要混合处理的物体。这是由于我们希望在混合运算开始之前，先将所有的非混合几何体都置于后台缓冲区中。然而，在这几种混合情况里其实并不需要对混合物体进行排序，因为这几种混合运算都满足交换律。也就是说，如果在初始时后台缓冲区中有一像素为颜色$\boldsymbol{B}$，则对该像素连续进行n次加法/减法/乘法单一混合运算，便无须考虑此间的计算顺序，即：

$$
\begin{aligned}
\boldsymbol{B}' &= \boldsymbol{B} + \boldsymbol{C}_0 + \boldsymbol{C}_1 + \ldots + \boldsymbol{C}_{n-1} \\
\boldsymbol{B}' &= \boldsymbol{B} - \boldsymbol{C}_0 - \boldsymbol{C}_1 - \ldots - \boldsymbol{C}_{n-1} \\
\boldsymbol{B}' &= \boldsymbol{B} \otimes \boldsymbol{C}_0 \otimes \boldsymbol{C}_1 \otimes \ldots \otimes \boldsymbol{C}_{n-1}
\end{aligned}
$$

10.5.5 混合与深度缓冲区

在使用加法/减法/乘法运算进行混合时，会涉及深度测试（depth test）这一问题。对于这个示例，我们仅用加法混合来讲解，但其中的思路也同样适用于减法/乘法的混合运算。如果要用加法混合来渲染一个物体集合S，并希望S中的物体不会互相遮挡，这就意味着我们只需将这些物体的颜色数据简单地累加即可（见图10.5）。为此，我们不愿在S中的物体之间进行深度测试。若开启深度测试，却并没有按从后至前

的顺序进行绘制，那么，当S中的两个物体存在遮挡关系，经过深度测试后，靠后的像素片段便会被丢弃，这意味着该物体的像素颜色将不会被累加至混合求和的结果之中。在渲染S中的物体时，我们可以通过禁止向深度缓冲区的写操作来禁用S中物体之间的深度测试。由于深度写入操作已被禁止，S中的物体在进行加法混合时，便不会将深度信息写入深度缓冲区，因此，S中的物体便不会因深度测试而直接覆盖其后的物体。注意，我们只是在绘制S（要用加法混合的方式来绘制的一个物体集合）中的物体时禁用了深度值写入操作，但深度值读取与深度检测仍然是开启的。这样一来，非混合几何体（比混合几何体先绘制的物体）仍将遮挡其后的混合几何体。比方说，如果我们有一个需要在墙后进行加法混合的物体集合，那么这些混合物体最后是看不到的，因为以非混合方式绘制的实心不透明墙体会挡住它们。至于如何禁用深度值写入操作以及设置深度检测，且看下章分解。



图10.5　若采用加法混合，则粒子叠加较多的点要亮于周围的其他点。随着粒子的扩散，原来的亮点逐渐变暗，而散至其他点处的粒子又与当地的粒子相叠加，使那些光点变得更亮

10.6 alpha通道

根据10.5.4节中的例子可以看出，源alpha分量能够用于在RGB混合的过程中控制像素的透明度。而混合方程中所用的源颜色实则来自于像素着色器。正如在第9章中所见，我们将漫反射材质的alpha值作为纹理着色器的alpha输出。这样一来，我们就能利用漫反射图（diffuse map）中的alpha通道来控制混合过程中的透明度。

```
float4 PS(VertexOut pin) : SV_Target
{
  float4 diffuseAlbedo = gDiffuseMap.Sample(
    gsamAnisotropicWrap, pin.TexC) * gDiffuseAlbedo;
  ...

  // 从漫反射反照率获取alpha值的常用方法
  litColor.a = diffuseAlbedo.a;
  return litColor;
}
```

我们往往可以在常见的图像编辑软件（例如Adobe Photoshop）中添加alpha通道，接着再将图像保存为支持alpha通道的格式，如DDS。

10.7　裁剪像素

有时候，我们希望彻底禁止某个源像素参与后续的处理。这可以通过HLSL的内置函数`clip(x)`来实现。此函数仅供像素着色器调用，若`x < 0`，则当前这一像素将从后面的处理阶段中丢弃。用这个函数来处理铁丝网纹理的绘制再合适不过了，例如，就如图10.6所示的效果。换句话说，用它来绘制透明与非透明相间的像素再好不过了。

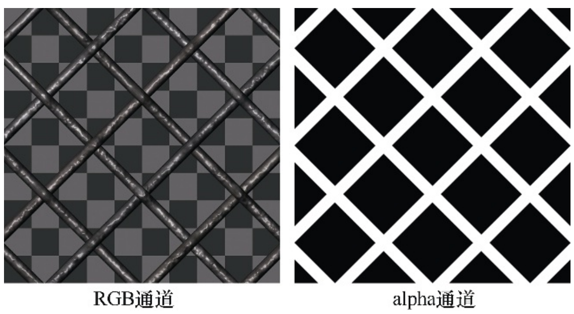


图10.6　具有alpha通道的铁丝网纹理。有着黑色alpha值的像素将被`clip`函数丢掉，从而不被绘制出来。因此，只有铁丝网部分会留存下来。从本质上来讲，alpha通道的用处在于从纹理中屏蔽掉非铁丝网的部分

在像素着色器中，我们将采集像素的alpha分量。如果该值极小接近于0，则表示此像素是完全透明的，那么我们就将此像素从后续处理中淘汰掉。

```
float4 PS(VertexOut pin) : SV_Target
{
  float4 diffuseAlbedo = gDiffuseMap.Sample(
    gsamAnisotropicWrap, pin.TexC) * gDiffuseAlbedo;

#ifdef ALPHA_TEST
  // 若alpha < 0.1则抛弃该像素。我们要在着色器中尽早执行此项测试，以尽快检测出满足条件的像素
```

```
  // 并退出着色器，从而跳过后续的相关处理过程
  clip(diffuseAlbedo.a - 0.1f);
#endif

  ...

  // 从漫反射反照率获取alpha值的常用手段
  litColor.a = diffuseAlbedo.a;

  return litColor;
}
```

可以观察到，只有在定义了ALPHA_TEST宏的时候才会实行透明像素的筛选。这是因为我们有时可能并不希望对某些渲染项执行clip方法，所以要有能力针对特殊的着色器开启或关闭对此函数的调用。而且alpha测试的开销也不小，因此只有在必要的情况下才使用它。

注意，通过混合操作也能实现相同的效果，但是使用clip函数更为有效。首先，它无须执行混合运算（即可以禁用混合操作）。其次，处理期间也不必考虑绘制顺序。此外，通过提前从像素着色器中抛弃像素，能够使之略过像素着色器的剩余指令（为终将被丢弃的像素执行这些后续指令是毫无意义的）。

注意 Note

纹理过滤操作可能会使alpha通道的数据略受影响，因此在裁剪像素时应对判断值留出适当的余地（即允许特定的误差）。例如，可以根据接近0的alpha值来裁剪像素，但不要按精确的0值进行处理。

图10.7所示的是“Blend Demo”（混合）演示程序的效果。它用透明混合的处理方法来绘制半透明的水，并通过`clip`测试来渲染铁丝网盒。另一个值得一提的变化是，由于当前立方体使用的是铁丝网

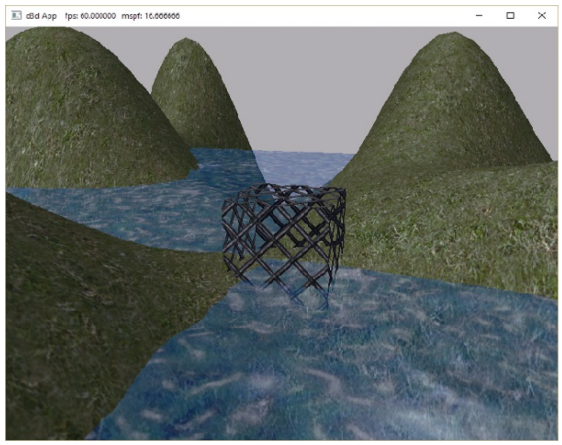


图10.7 “Blend Demo”演示程序的效果

纹理，因此我们就应对alpha测试物体都禁用背面剔除（不然后面就穿帮了！）：

```
// 针对alpha测试物体所采用的PSO
D3D12_GRAPHICS_PIPELINE_STATE_DESC alphaTestedPsoDesc = opaquePsoDesc;
alphaTestedPsoDesc.PS =
{
  reinterpret_cast<BYTE*>(mShaders["alphaTestedPS"]->GetBufferPointer()),
    mShaders["alphaTestedPS"]->GetBufferSize()
};
alphaTestedPsoDesc.RasterizerState.CullMode = D3D12_CULL_MODE_NONE;
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(
  &alphaTestedPsoDesc, IID_PPV_ARGS(&mPSOs["alphaTested"])));
```

10.8　雾

为了在游戏中模拟出一些特定的天气状况，我们往往需要实现雾化效果，如图10.8所示。除了雾效本身这个主要目的之外，它还有一些其他的用处。例如，浓雾可以掩饰远处景物在渲染上的失真，以及防止发生物体突然出现（popping）[1]的情况。突然出现是指一个物体原本位于视锥体远平面的后侧，但由于摄像机的移动，使之突然出现在视锥体的范围之内，并因此令它变得可见，以致该物体看起来似乎是突然“瞬移”到了场景之中。在远处设立一层雾气便可掩盖物体突然出现的现象。注意，如果场景处于晴朗的白天，不妨在远处设有少量的薄雾。因为就算是万里无云的好天气，像高山这种远处的景物依旧云雾缭绕，且雾气将随着深度函数值的增加而逐渐变浓（失去对比度，lose contrast）。此时，我们便可以通过雾效来模拟这种大气透视的现象。

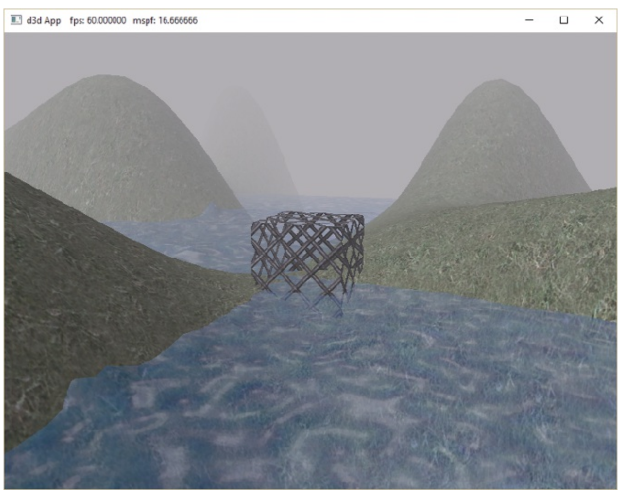


图10.8　开启雾效的“Blend Demo”例程的效果

实现雾化效果的流程如下：如图10.9所示，首先指明雾的颜色、由

摄像机到雾气的最近距离以及雾的分散范围（即从雾到摄像机的最近距离至雾能完全覆盖物体的这段范围），接下来再将网格三角形上点的颜色置为原色与雾色的加权平均值：

$$\begin{aligned}\text{foggedColor} &= \text{litColor} + \text{s}(\text{fogColor} - \text{litColor}) \\ &= (1-\text{s})\text{cdotlitColor} + \text{s}\cdot\text{fogColor}\end{aligned}$$

参数s的范围为[0, 1]，由一个以摄像机位置与被雾覆盖物体表面点之间的距离作为参数的函数来确定。随着该表面点与观察点之间距离的增加，它会被雾气遮挡得愈加朦胧。参数s的定义如下：

$$s = \text{saturate}\left(\frac{\text{dist}(\boldsymbol{p},\boldsymbol{E}) - fogStart}{fogRange}\right)$$

其中，dist($\boldsymbol{p}$, $\boldsymbol{E}$)为表面点$\boldsymbol{p}$与摄像机位置$\boldsymbol{E}$之间的距离。而函数saturate会将其参数限制在区间[0, 1]内：

$$\text{saturate}(x) = \begin{cases} x,\ 0 \leqslant x \leqslant 1 \\ 0,\ x < 0 \\ 1,\ x > 1 \end{cases}$$

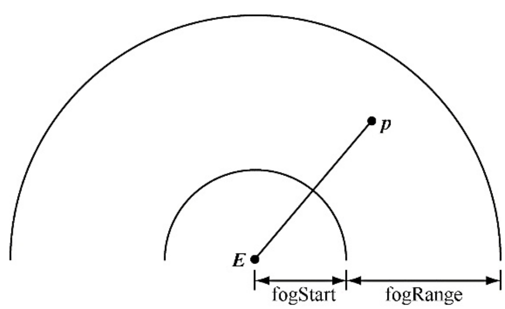


图10.9　摄像机$\boldsymbol{E}$到某点$\boldsymbol{P}$的距离，fogStart（摄像机到雾气的最近距离）与fogRange（雾气的范围）即是相关参数

图10.10所示为根据距离函数而绘制的s图像。由此可以看出，当

$(\boldsymbol{p}, \boldsymbol{E}) \leqslant \text{fogStart}$时$s=0$，而雾的颜色则由下式给出：

foggedColor = litColor

换句话说，当物体表面点到摄像机的距离小于fogStart时，雾色就不会改变物体顶点的本色。顾名思义，只有表面点到摄像机的距离至少为“fogStart”（雾效开始）时，其颜色才会受到雾色的影响。

设fogEnd = fogStart + fogRange。当$\text{dist}(\boldsymbol{p}, \boldsymbol{E}) \geqslant \text{fogEnd}$时$s=1$，且雾色为：

foggedColor = fogColor

这便是说，当物体表面点的位置到观察点的距离大于或等于fogEnd时，浓雾会将它完全遮住——所以我们只能看到雾气的颜色。

从图10.10中不难看出，当$\text{fogStart} < \text{dist}(\boldsymbol{p}, \boldsymbol{E}) < \text{fogEnd}$时，随着$\text{dist}(\boldsymbol{p}, \boldsymbol{E})$从fogStart向fogEnd递增，变量$s$也呈线性地由0增加至1。这表明随着距离的增加，雾色会越来越浓重，而物体原色也愈加寡淡。这是显而易见的，因为随着距离的增加，雾气势必越发浓重，以致越远的景物越迷蒙。

下列着色器代码展示了如何来实现雾效。我们先计算距离，并在像素层级进行插值，最后再求出光照颜色。

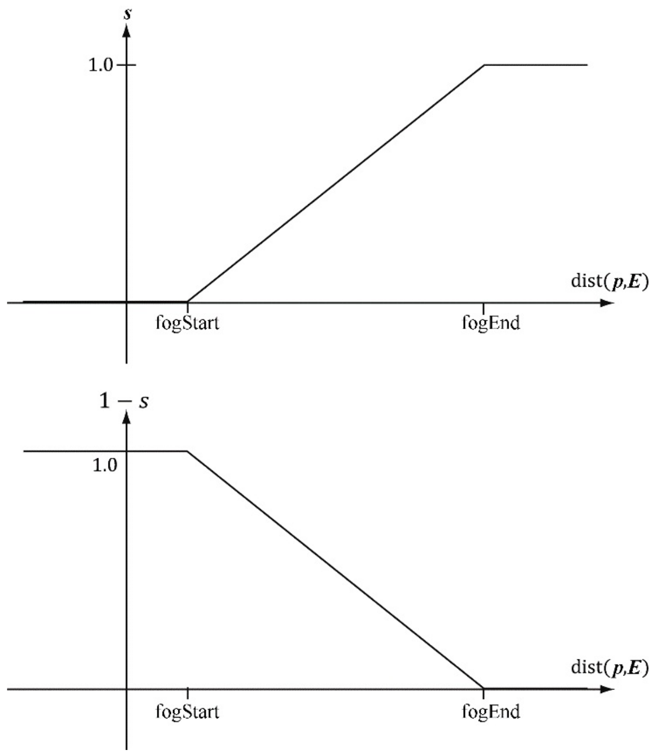


图10.10　上图中，根据距离函数而得到的s（雾气的颜色权值）的图像。下图中，根据距离函数而得到的$1-s$（物体的颜色权值）的图像。随着s的增加，($1-s$)势必将相应地减少

```
// 光源数量的默认值
#ifndef NUM_DIR_LIGHTS
  #define NUM_DIR_LIGHTS 3
#endif

#ifndef NUM_POINT_LIGHTS
  #define NUM_POINT_LIGHTS 0
#endif

#ifndef NUM_SPOT_LIGHTS
  #define NUM_SPOT_LIGHTS 0
#endif

// 包含光照所用的结构体与函数
#include "LightingUtil.hlsl"

Texture2D  gDiffuseMap : register(t0);

SamplerState gsamPointWrap    : register(s0);
SamplerState gsamPointClamp    : register(s1);
SamplerState gsamLinearWrap    : register(s2);
```

```
SamplerState gsamLinearClamp   : register(s3);
SamplerState gsamAnisotropicWrap : register(s4);
SamplerState gsamAnisotropicClamp : register(s5);

// 每一帧中都在变化的常量数据
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorld;
  float4x4 gTexTransform;
};

// 绘制过程中所用的杂项常量数据
cbuffer cbPass : register(b1)
{
  float4x4 gView;
  float4x4 gInvView;
  float4x4 gProj;
  float4x4 gInvProj;
  float4x4 gViewProj;
  float4x4 gInvViewProj;
  float3 gEyePosW;
  float cbPerObjectPad1;
  float2 gRenderTargetSize;
  float2 gInvRenderTargetSize;
  float gNearZ;
  float gFarZ;
  float gTotalTime;
  float gDeltaTime;
  float4 gAmbientLight;

  // 允许应用程序在每一帧都能改变雾效参数
  // 例如，我们可能只在一天中的特定时间才使用雾效
  float4 gFogColor
  float gFogStart;
  float gFogRange;
  float2 cbPerPassPad2;

  // 对于每个以MaxLights 为光源数量最大值的对象而言，索引[0,NUM_DIR_LIGHTS)表
示的是方向光
  // 源，索引[NUM_DIR_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHTS)表示的是点光源
  // 索引[NUM_DIR_LIGHTS+NUM_POINT_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHT+
NUM_SPOT_
  // LIGHTS)表示的是聚光灯光源

  Light gLights[MaxLights];
};
```

```
// 每种材质中的不同常量数据
cbuffer cbMaterial : register(b2)
{
  float4   gDiffuseAlbedo;
  float3   gFresnelR0;
  float    gRoughness;
  float4x4 gMatTransform;
};

struct VertexIn
{
  float3 PosL    : POSITION;
  float3 NormalL : NORMAL;
  float2 TexC    : TEXCOORD;
};

struct VertexOut
{
  float4 PosH    : SV_POSITION;
  float3 PosW    : POSITION;
  float3 NormalW : NORMAL;
  float2 TexC    : TEXCOORD;
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout = (VertexOut)0.0f;

  // 把顶点变换到世界空间
  float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
  vout.PosW = posW.xyz;

  // 假设要进行的是等比缩放，否则便应使用世界矩阵的逆转置矩阵
  vout.NormalW = mul(vin.NormalL, (float3x3)gWorld);

  // 将顶点变换到齐次裁剪空间
  vout.PosH = mul(posW, gViewProj);

  // 为三角形插值而输出顶点属性
  float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
  vout.TexC = mul(texC, gMatTransform).xy;

  return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
```

```
  float4 diffuseAlbedo = gDiffuseMap.Sample(
    gsamAnisotropicWrap, pin.TexC) * gDiffuseAlbedo;

#ifdef ALPHA_TEST
  // 如果纹理的alpha < 0.1则丢弃该像素。我们应尽早执行这项测试，以便尽快检测到满足条件的像素
  // 并提前退出着色器，从而跳过着色器的后续处理
  clip(diffuseAlbedo.a - 0.1f);
#endif

  // 对法线插值可能导致其非规范化，因而要重新对它进行规范化处理
  pin.NormalW = normalize(pin.NormalW);

  // 光线经表面上一点反射到观察点这一方向上的向量
  float3 toEyeW = gEyePosW - pin.PosW;
  float distToEye = length(toEyeW);
  toEyeW /= distToEye; // 规范化处理

  // 光照项
  float4 ambient = gAmbientLight*diffuseAlbedo;

  const float shininess = 1.0f - gRoughness;
  Material mat = { diffuseAlbedo, gFresnelR0, shininess };
  float3 shadowFactor = 1.0f;
  float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
    pin.NormalW, toEyeW, shadowFactor);

  float4 litColor = ambient + directLight;
#ifdef FOG
  float fogAmount = saturate((distToEye - gFogStart) / gFogRange);
  litColor = lerp(litColor, gFogColor, fogAmount);
#endif

  // 从漫反射反照率中获取alpha值的常用手法
  litColor.a = diffuseAlbedo.a;

  return litColor;
}
```

有一些场景可能并不会用到雾效，因此我们将该功能设为可选项，若要使用，则需在编译着色器时定义FOG宏。这样一来，如果不希望使用雾效，则不必为相关运算而付出开销代价。在演示程序中，我们通过向CompileShader函数提供下列D3D_SHADER_MACRO结构体来开启雾

效：

```
const D3D_SHADER_MACRO defines[] =
{
  "FOG", "1",
  NULL, NULL
};

mShaders["opaquePS"] = d3dUtil::CompileShader(
  L"Shaders\\Default.hlsl", defines, "PS", "ps_5_0");
```

注意 Note

通过观察可以发现，我们不仅在雾效运算中使用了distToEye，而且在对法线进行规范化处理时也用到了此值。这里再给出一个优化方面稍差的实现，试对比：

```
float3 toEyeW = normalize(gEyePosW - pin.PosW);
float distToEye = distance(gEyePosW, pin.PosW);
```

这段代码其实将向量toEyeW的长度计算了两次：一次是用normalize函数，另一次是用distance函数。

10.9　小结

1．混合是一种将当前待光栅化的像素（即源像素）与之前已光栅化并存至后台缓冲区的像素（即目标像素）相融合（组合）的技术。它使我们可以渲染出半透明效果的物体，如水与玻璃。

2．混合方程形如：

$$\boldsymbol{C} = \boldsymbol{C}_{\text{src}} \otimes \boldsymbol{F}_{\text{src}} \boxplus \boldsymbol{C}_{\text{dst}} \otimes \boldsymbol{F}_{\text{dst}}$$

$$A = A_{\text{src}} F_{\text{src}} \boxplus A_{\text{dst}} F_{\text{dst}}$$

注意到，RGB分量与alpha分量的混合运算实际上是各自单独展开的。二元运算符⊞是枚举类型`D3D12_BLEND_OP`的成员之一。

3．$\boldsymbol{F}_{\text{src}}$、$\boldsymbol{F}_{\text{dst}}$、$F_{\text{src}}$与$F_{\text{dst}}$都被称为混合因子，它们是混合方程可自定义的法宝。这些因子是枚举类型`D3D12_BLEND`中的成员之一。另外，对于alpha混合方程来说，不能采用以`_COLOR`作为后缀的混合因子。

4．源alpha的信息来自于漫反射材质。在我们自己编写的应用框架中，漫反射材质由纹理图来定义，而所需的alpha信息正是存于纹理的alpha通道。

5．通过HLSL的内部函数`clip(x)`可以将源像素从后续的处理过程中完全屏蔽掉。此函数仅可在像素着色器中调用，当`x < 0`时它便丢弃当前的像素。此外，该函数对不透明与透明相间像素的绘制也是极为有效的（即此函数可用于过滤完全透明的像素，也就是alpha值接近于0的

像素）。

6．可以使用雾效来模拟各种气象环境效果和大气透视现象，以此来掩饰远处场景渲染的失真以及物体突然出现于视锥体之中闯进用户视野内的情况。在我们所用的线性雾效模型中，需要指定雾气的颜色，从摄像机至雾效的最近距离以及雾的出现范围。此时，网格三角形上某点的颜色是其原色与雾色的加权平均值：

$$\begin{aligned}\text{foggedColor} &= l\text{itColor} + s(\text{fogColor} - \text{litColor}) \\ &= (1 - s) \cdot \text{litColor} + s \cdot \text{fogColor}\end{aligned}$$

参数s的取值范围是[0, 1]，由一个以摄像机位置与物体表面点之间距离为参数的函数来表示。随着表面点与观察点之间距离的增加，表面点受雾气的影响将变得看起来越来越朦胧。

10.10 练习

1．体会不同的混合运算与不同混合因子的组合效果。

2．修改“Blend Demo”演示程序，首先绘制水流。请解释这一改变所产生的效果。

3．设fogStart = 10且fogRange = 200。针对下列各种情况来计算foggedColor。

（a）$\mathrm{dist}(\boldsymbol{p}, \boldsymbol{E}) = 160$

（b）$\mathrm{dist}(\boldsymbol{p}, \boldsymbol{E}) = 110$

（c）$\mathrm{dist}(\boldsymbol{p}, \boldsymbol{E}) = 60$

（d）$\mathrm{dist}(\boldsymbol{p}, \boldsymbol{E}) = 30$

4．通过查看生成的着色器汇编代码来验证：当没有定义`ALPHA_TEST`宏时，编译后的像素着色器不会用到`discard`指令，反之则会用到。此`discard`指令对应于HLSL中的`clip`指令。

5．修改“Blend Demo”演示程序，创建并应用禁止向红色通道和绿色通道写入颜色信息的混合渲染状态。

---

[1] 这个词也常用于表示不同LOD（level of detail，细节层级）模型突

兀的切换所造成的视觉上的跳跃，此时常译作“突跃”。这与当前文中的popping是两种概念。

第11章　模板

模板缓冲区（stencil buffer）是一种“离屏”（off-screen）缓冲区，我们可以利用它来实现一些特殊效果。模板缓冲区、后台缓冲区以及深度缓冲区都有着相同的分辨率，这样一来，这三者相同位置上的像素就能一一对应起来。回顾4.1.5节可知，在指定一个模板缓冲区时，要将它与一个深度缓冲区配合使用。顾名思义，这种缓冲区所起到的作用就如同印刷过程中所用的模板一样，我们可以用它来阻止特定的像素片段渲染至后台缓冲区中。

举个例子，当渲染一面镜子时，我们需要将物体反映到镜子所在的平面上。当然，只应绘制出镜子中的镜像部分。这时，我们就能通过模板缓冲区来阻止镜子范围以外镜像部分的绘制操作（见图11.1）。

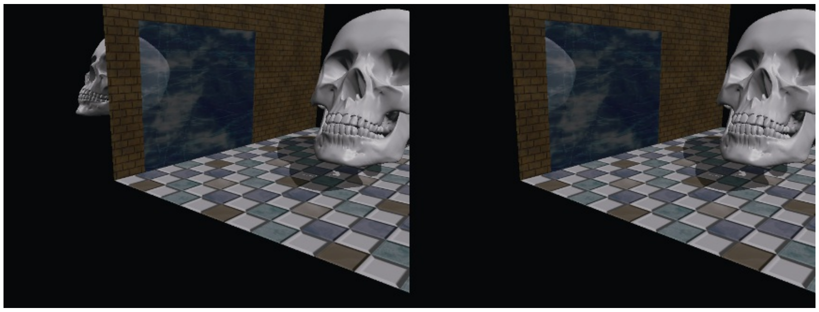


图11.1 （左图）在镜子中正确地反射出了骷髅头的映像。由于深度测试的原因，使得砖块后的骷髅头部分没有显现出来。但是，我们仍然能够从左侧看到墙面后露出的骷髅头镜像，显然，这是违反3D视觉原理的（镜像只能在镜中呈现）。（右图）借助模板缓冲区，我们便能阻止镜面外映像的绘制

要设置模板缓冲区（以及深度缓冲区）状态，就需填写D3D12_DEPTH_STENCIL_DESC结构体实例，并将其赋予流水线状态对象（PSO）的D3D12_GRAPHICS_PIPELINE_STATE_DESC::DepthStencilState字段。学习模板缓冲区用法的最佳方式即从现存的示例应用程序着手。一旦对实例中的模板缓冲区部分有了感性的认识，我们便可以更加得心应手地运用它了。

学习目标：

1．探究如何通过填写流水线状态对象中的D3D12_DEPTH_STENCIL_DESCDepthStencilState字段来控制深度缓冲区以及模板缓冲区。

2．学习通过模板缓冲区来防止镜像被绘至镜子以外的区域，以此来实现正确的镜像效果。

3．了解双重混合（double blending）的机制，从而利用模板缓冲区来有效地杜绝这一情况的发生。

4．知晓深度复杂性（depth complexity）的概念，并介绍两种方法来度量场景的深度复杂性。

11.1 深度/模板缓冲区的格式及其资源数据的清理

回顾前文的内容便会想起，深度/模板缓冲区其实也是一种纹理，因而必须用下列特定的数据格式来创建它。深度/模板缓冲可用的格式如下：

1. DXGI_FORMAT_D32_FLOAT_S8X24_UINT：此格式用一个32位浮点数来指定深度缓冲区，并以另一个32位无符号整数来指定模板缓冲区。其中，无符号整数里的8位用于将模板缓冲区映射到范围[0, 255]，另外24位不可用，仅作填充占位。

2. DXGI_FORMAT_D24_UNORM_S8_UINT：指定一个无符号的24位深度缓冲区，并将其映射到范围[0, 1]内。另外8位（无符号整数）用于令模板缓冲区映射至范围[0, 255]。

在D3DApp应用框架中，当要创建深度缓冲区时就要像下面那样来指定它的格式：

```
DXGI_FORMAT mDepthStencilFormat = DXGI_FORMAT_D24_UNORM_S8_UINT;
depthStencilDesc.Format = mDepthStencilFormat;
```

我们可以在绘制每一帧画面之初，用以下方法来重置模板缓冲区中的局部数据（也可用于清理深度缓冲区）。

```
void ID3D12GraphicsCommandList::ClearDepthStencilView(
  D3D12_CPU_DESCRIPTOR_HANDLE DepthStencilView,
```

```
  D3D12_CLEAR_FLAGS ClearFlags,
  FLOAT Depth,
  UINT8 Stencil,
  UINT NumRects,
  const D3D12_RECT *pRects);
```

1. DepthStencilView：待清理的深度/模板缓冲区视图的描述符。

2. ClearFlags：指定为D3D12_CLEAR_FLAG_DEPTH仅清理深度缓冲区，指定为D3D12_CLEAR_FLAG_STENCIL只清理模板缓冲区，指定为D3D12_CLEAR_FLAG_DEPTH | D3D12_CLEAR_FLAG_STENCIL则同时清理这两种缓冲区。

3. Depth：将此浮点值设置到深度缓冲区中的每一个像素。此浮点数x务必满足$0 \leqslant x \leqslant 1$。

4. Stencil：将此整数值设置到模板缓冲区中的每一个像素。此整数n必须满足$0 \leqslant n \leqslant 255$。

5. NumRects：数组pRects中所指引的矩形数量。

6. pRects：一个D3D12_RECT类型数组，它标定了一系列深度/模板缓冲区内要清理的区域。若指定为nullptr则清理整个深度/模板缓冲区。

我们在演示程序中的每一帧都已经调用此方法，形如：

```
mCommandList->ClearDepthStencilView(DepthStencilView(),
  D3D12_CLEAR_FLAG_DEPTH | D3D12_CLEAR_FLAG_STENCIL,
```

```
1.0f, 0, 0, nullptr);
```

11.2　模板测试

如前所述，我们可以通过模板缓冲区来阻止对后台缓冲区特定区域的绘制行为。而这项操作实则是由模板测试（stencil test）来决定的，它的处理过程如下：

```
if( StencilRef & StencilReadMask ⊴ Value & StencilReadMask )
```

```
  accept pixel
else
  reject pixel
```

模板测试会随着像素的光栅化过程而执行（即在输出合并阶段进行）。若模板功能呈开启状态，则需经过下面两处运算。

1．左运算数（left-hand-side，LHS）由程序中定义的模板参考值（stencil reference value）`StencilRef`与程序内定义的掩码值（masking value）`StencilReadMask`通过AND（与）运算来加以确定。

2．右运算数（right-hand-side，RHS）由正在接受模板测试的特定像素位于模板缓冲区中的对应值`Value`与程序中定义的掩码值`StencilReadMask`经过AND计算来加以确定。

可以发现，左运算数与右运算数中的`StencilReadMask`是同一个值。接下来，模板测试用程序中所选定的比较函数（comparison function）⊴对左运算数与右运算数进行比对，从而得到布尔类型的返回值。如果测试结果为true，就将当前受检测的像素写入后台缓冲区（即

假设此像素已通过深度测试）；如果测试结果为false，则禁止此像素向后台缓冲区的写操作。当然，如果一个像素因模板测试失败而被丢弃，它的相关数据也不会被写入深度缓冲区。

运算符“⊴”是D3D12_COMPARISON_FUNC枚举类型所定义的比较函数之一[1]：

```
typedef enum D3D12_COMPARISON_FUNC
{
  D3D12_COMPARISON_FUNC_NEVER = 1,
  D3D12_COMPARISON_FUNC_LESS = 2,
  D3D12_COMPARISON_FUNC_EQUAL = 3,
  D3D12_COMPARISON_FUNC_LESS_EQUAL = 4,
  D3D12_COMPARISON_FUNC_GREATER = 5,
  D3D12_COMPARISON_FUNC_NOT_EQUAL = 6,
  D3D12_COMPARISON_FUNC_GREATER_EQUAL = 7,
  D3D12_COMPARISON_FUNC_ALWAYS = 8,
} D3D12_COMPARISON_FUNC;
```

1. D3D12_COMPARISON_FUNC_NEVER：该函数总是返回false。

2. D3D12_COMPARISON_FUNC_LESS：用运算符<替换⊴。

3. D3D12_COMPARISON_FUNC_EQUAL：用运算符==替换⊴。

4. D3D12_COMPARISON_FUNC_LESS_EQUAL：用运算符≤替换⊴。

5. D3D12_COMPARISON_FUNC_GREATER：用运算符>替换⊴。

6. D3D12_COMPARISON_FUNC_NOT_EQUAL：用运算符!=替换⊴。

7. D3D12_COMPARISON_FUNC_GREATER_EQUAL：用运算符≥替换⊴

。

8. D3D12_COMPARISON_FUNC_ALWAYS：此函数总是返回true。

11.3　描述深度/模板状态

要描述深度/模板状态，就需填写D3D12_DEPTH_STENCIL_DESC实例：

```
typedef struct D3D12_DEPTH_STENCIL_DESC {
  BOOL DepthEnable; // 默认值为True

  // 默认值为 D3D12_DEPTH_WRITE_MASK_ALL
  D3D12_DEPTH_WRITE_MASK DepthWriteMask;

  // 默认值为 D3D12_COMPARISON_LESS
  D3D12_COMPARISON_FUNC DepthFunc;

  BOOL StencilEnable;        // 默认值为 False
  UINT8 StencilReadMask;     // 默认值为 0xff，即D3D12_DEFAULT_STENCIL_WRIT
E_MASK
  UINT8 StencilWriteMask;    // 默认值为 0xff，即D3D12_DEFAULT_STENCIL_WRIT
E_MASK
  D3D12_DEPTH_STENCILOP_DESC FrontFace;
  D3D12_DEPTH_STENCILOP_DESC BackFace;
} D3D12_DEPTH_STENCIL_DESC;
```

11.3.1　深度信息的相关设置

深度信息的相关设置如下。

1．DepthEnable：设置为true，则开启深度缓冲；设置为false，则禁用。当深度测试被禁止时，物体的绘制顺序就变得极为重要，否则位于遮挡物之后的像素片段也将被绘制出来（回顾4.1.5节）。如果深度缓冲被禁用，则深度缓冲区中的元素便不会被更新，DepthWriteMask项的设置也不会起作用。

2. DepthWriteMask：可将此参数设置为D3D12_DEPTH_WRITE_MASK_ZERO或者D3D12_DEPTH_WRITE_MASK_ALL，但两者不能共存。假设DepthEnable为 true，若把此参数设置为D3D12_DEPTH_WRITE_MASK_ZERO便会禁止对深度缓冲区的写操作，但仍可执行深度测试；若将该项设为D3D12_DEPTH_WRITE_MASK_ALL，则通过深度测试与模板测试的深度数据将被写入深度缓冲区。这种控制深度数据读写的能力，为某些特效的实现提供了良好的契机。

3. DepthFunc：将该参数指定为枚举类型D3D12_COMPARISON_FUNC的成员之一，以此来定义深度测试所用的比较函数。此项一般被设为D3D12_COMPARISON_FUNC_LESS，因而常常执行如4.1.5节中所述的深度测试。即，若给定像素片段的深度值小于位于深度缓冲区中对应像素的深度值，则接受该像素片段（离摄像机近的物体遮挡距摄像机远的物体）。当然，也正如我们所看到的，Direct3D也允许用户根据需求来自定义深度测试。

11.3.2　模板信息的相关设置

模板信息的相关设置如下。

1. StencilEnable：设置为true，则开启模板测试；设置为false，则禁用。

2．StencilReadMask：该项用于以下模板测试：

```
if( StencilRef & StencilReadMask ⊴ Value & StencilReadMask )
```

```
  accept pixel
else
  reject pixel
```

若采用该项的默认值，则不会屏蔽任何一位模板值。

```
#define      D3D12_DEFAULT_STENCIL_READ_MASK  ( 0xff )
```

3．StencilWriteMask：当模板缓冲区被更新时，我们可以通过写掩码（write mask）来屏蔽特定位的写入操作。例如，如果我们希望防止前4位数据被改写，便可以将写掩码设置为0x0f。而默认配置是不会屏蔽任何一位模板值的：

```
#define D3D12_DEFAULT_STENCIL_WRITE_MASK ( 0xff )
```

4．FrontFace：填写一个D3D12_DEPTH_STENCILOP_DESC结构体实例，以指示根据模板测试与深度测试的结果，应对正面朝向的三角形要进行何种模板运算。

5．BackFace：填写一个D3D12_DEPTH_STENCILOP_DESC结构体实例，以指出根据模板测试与深度测试的结果，应对背面朝向的三角形要进行何种模板运算。

```
typedef struct D3D12_DEPTH_STENCILOP_DESC {
 D3D12_STENCIL_OP StencilFailOp;      // 默认值为: D3D12_STENCIL_OP_KEEP
 D3D12_STENCIL_OP StencilDepthFailOp; // 默认值为: D3D12_STENCIL_OP_KEEP
 D3D12_STENCIL_OP StencilPassOp;      // 默认值为: D3D12_STENCIL_OP_KEEP
 D3D12_COMPARISON_FUNC StencilFunc;   // 默认值为: D3D12_COMPARISON_FUNC_AL
```

```
WAYS
} D3D12_DEPTH_STENCILOP_DESC;
```

1. StencilFailOp：枚举类型D3D12_STENCIL_OP中的成员之一，描述了当像素片段在模板测试失败时，应该怎样更新模板缓冲区。

2. StencilDepthFailOp：枚举类型D3D12_STENCIL_OP中的成员之一，描述了当像素片段通过模板测试，却在深度测试失败时，应如何更新模板缓冲区。

3. StencilPassOp：枚举类型D3D12_STENCIL_OP中的成员之一，描述了当像素片段通过模板测试与深度测试时，该怎样更新模板缓冲区。

4. StencilFunc：枚举类型D3D12_COMPARISON_FUNC中的成员之一，定义了模板测试所用的比较函数。

```
typedef
enum D3D12_STENCIL_OP
{
  D3D12_STENCIL_OP_KEEP      = 1,
  D3D12_STENCIL_OP_ZERO      = 2,
  D3D12_STENCIL_OP_REPLACE   = 3,
  D3D12_STENCIL_OP_INCR_SAT  = 4,
  D3D12_STENCIL_OP_DECR_SAT  = 5,
  D3D12_STENCIL_OP_INVERT    = 6,
  D3D12_STENCIL_OP_INCR      = 7,
  D3D12_STENCIL_OP_DECR      = 8
} D3D12_STENCIL_OP;
```

1. D3D12_STENCIL_OP_KEEP：不修改模板缓冲区，即保持当前的数据。

2. D3D12_STENCIL_OP_ZERO：将模板缓冲区中的元素设置为0。

3. D3D12_STENCIL_OP_REPLACE：将模板缓冲区中的元素替换为用于模板测试的模板参考值（StencilRef）。注意，只有当我们将深度/模板缓冲区状态块绑定到渲染流水线时，才能够设定StencilRef值（见11.3.3节）。

4. D3D12_STENCIL_OP_INCR_SAT：对模板缓冲区中的元素进行递增（increment）操作。如果递增值超出最大值（例如，8位模板缓冲区的最大值为255），则将此模板缓冲区元素限定为最大值。

5. D3D12_STENCIL_OP_DECR_SAT：对模板缓冲区中的元素进行递减（decrement）操作。如果递减值小于0，则将该模板缓冲区元素限定为0。

6. D3D12_STENCIL_OP_INVERT：对模板缓冲区中的元素数据按二进制位进行反转。

7. D3D12_STENCIL_OP_INCR：对模板缓冲区中的元素进行递增操作。如果递增值超出最大值（例如，对于8位模板缓冲区而言，其最大值为255），则环回至0。

8. D3D12_STENCIL_OP_DECR：对模板缓冲区中的元素进行递减操作。如果递减值小于0，则环回至可取到的最大值。

注意 Note

通过观察能够看出，对正面朝向三角形与背面朝向三角形所进行的模板运算可以是互不相同的。由于在执行背面剔除后背面朝向的多边形并不会得到渲染，所以在这种情况下对BackFace的设置便是无足轻重的。然而，我们有时候却需要针对特定的图形学算法或透明几何体（例如铁丝网盒，我们能透过它看到其背后的面）的处理来渲染背面朝向的多边形。而此时对BackFace的设置则又变得特别重要。

11.3.3　创建和绑定深度/模板状态

一旦将描述深度/模板状态的D3D12_DEPTH_STENCIL_DESC实例填写完整，我们就可以将其赋予PSO的D3D12_GRAPHICS_PIPELINE_STATE_DESC::DepthStencilState字段。而使用此PSO绘制的几何体，都将根据上述的深度/模板设置来进行渲染。

有一个细节我们还未曾提及，即如何来设置模板参考值。此操作可由ID3D12GraphicsCommandList:: OMSetStencilRef方法来实现，它以一个无符号整数作为参数。例如，下列代码将模板参考值设置为1：

```
mCommandList->OMSetStencilRef(1);
```

11.4　实现平面镜效果

在现实生活中，许多物体的表面皆能看作镜面，我们可以从中看到物体的镜像。本节将描述如何在3D应用程序中模拟镜面效果。注意，为了方便起见，此处将实现镜面的任务简化为仅完成平面镜效果。比如说，一辆锃光瓦亮的小轿车会反映出周围环境的镜像，但车体光滑、具有弧线，却非平面。我们要渲染的镜像类似于光滑的大理石地板或悬挂在墙上的镜子所反映出的镜像——换句话说，我们要模拟的是位于平面上的镜面。

在实现镜像编程的过程中亟需解决两个问题。首先，我们必须了解任意平面反射物体的相关原理，以此来正确地绘制镜像。其次，我们一定要将镜像显示在镜子当中，即必须以某种方式“标记”出表面内的镜面部分。而后，随着渲染工作的开展，只有处于镜面内的物体映像部分才会被绘制出来。对此可以回顾图11.1，那是我们第一次提及这个概念的地方。

第一个问题通过一些解析几何学的知识便可以轻松地解决，相关的讨论参见附录C。第二个问题则能够用模板缓冲区来解决。

11.4.1　镜像概述

注意 Note

在绘制镜像时，我们也需要将光源反映到镜子所在的平面内，否则会造成镜像中的光照不够精准。

图11.2展示了一个物体镜像的绘制过程，我们只需将它反射到镜面的背面即可。可是这样一来却引入了图11.1所示的问题，即物体（这个示例中是骷髅头）的镜像仅仅是场景中的另一个“实物”而已，如果没有其他东西遮挡，就能直接看到它。然而，现实中的镜像却只有在镜子中才能看到。要解决这个问题，就要用到模板缓冲区技术，借此即可阻止后台缓冲区中特定区域的渲染操作。因此，超出镜面范围的骷髅头镜像绘制操作便会被模板缓冲区制止。下面所列的是实现该效果的步骤要点。

1．将地板、墙壁以及骷髅头实物照常渲染到后台缓冲区内（不包括镜子）。注意，此步骤不修改模板缓冲区。

2．清理模板缓冲区，将其整体置零。图11.3展示了此时的后台缓冲区与模板缓冲区中的情况（为了简单起见，这里用立方体替代骷髅头实物）。

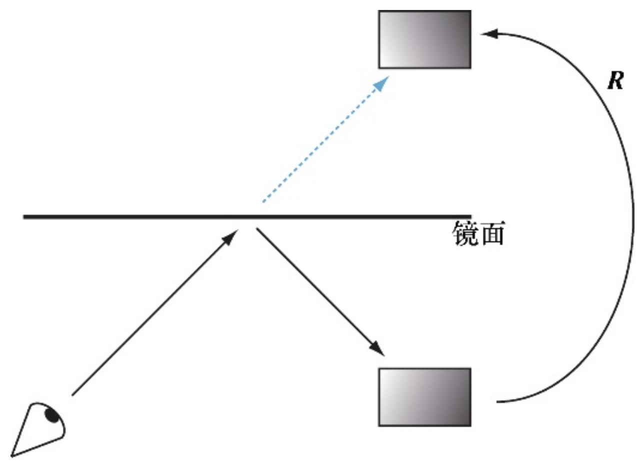


图11.2　图中展示的是观察者在镜面中查看盒子镜像的原理。为了模拟这个场景，
我们会在镜面背后反射出盒子的镜像，而在镜面外侧则像以往那样渲染盒体实物

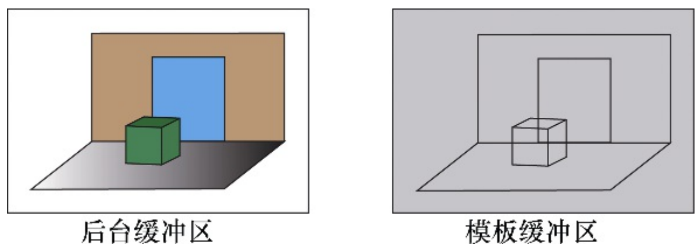


图11.3　将地板、墙壁与骷髅头都绘制到后台缓冲区中，并将模板缓冲区清理为0（用浅灰色来表示）。
绘制在模板缓冲区中的黑色轮廓线条反映的是：后台缓冲区与模板缓冲区中像素之间的对照关系，
而并非模板缓冲区中所绘的实际数据

3．仅将镜面渲染到模板缓冲区中。若要禁止其他颜色数据写入到后台缓冲区，可用下列设置所创建的混合状态：

```
D3D12_RENDER_TARGET_BLEND_DESC::RenderTargetWriteMask = 0;
```

再通过以下配置来禁止向深度缓冲区的写操作：

```
D3D12_DEPTH_STENCIL_DESC::DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ZERO;
```

在向模板缓冲区渲染镜面的时候，我们将模板测试设置为每次都成功（D3D12_COMPARISON_ FUNC_ALWAYS），并且在通过测试时用1（StencilRef模板参考值）来替换（通过D3D12_STENCIL_OP_REPLACE来设置）模板缓冲区元素。如果深度测试失败（这是有可能发生的，例如当骷髅头遮住部分镜子的时候），则应当采用枚举项D3D12_STENCIL_OP_KEEP，使模板缓冲区中的对应像素保持不变。由于仅向模板缓冲区绘制了镜面，因此在模板缓冲区内，除了镜面可见部分的对应像素为1，其他像素皆为0。图11.4所示的即为更新后的模板缓冲区。换言之，我们其实就是在模板缓冲区中标记了镜面的可见像素而已。

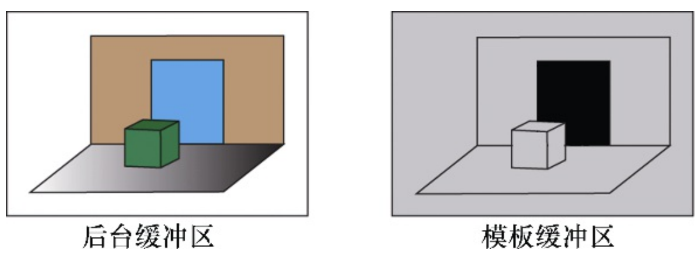


图11.4　把镜面渲染到模板缓冲区中，其实就是在模板缓冲区中标记出镜面可视部分的对应像素。模板缓冲区中实心黑色区域的模板元素取值为1。但请注意，由于被立方体挡住部分的深度测试会失败，所以在模板缓冲区中的这一范围内，元素的取值并不为1（立方体与黑色镜面重合的部分，也就是立方体位于镜面前方的这一部分）

注意　Note

保证先绘制骷髅头实物，后将镜面渲染至模板缓冲区的顺序是很重要的。这样一来，深度测试的失败会令镜面的像素被骷髅头实物的像素

所遮挡，因而也就不必再对模板缓冲区进行二次修改了。我们并不希望把模板缓冲区中镜面被遮挡部分的值设为1，那样将导致在骷髅头实物位于镜面前方的范围内也能显示出镜面内容。

4．现在我们来将骷髅头的镜像渲染至后台缓冲区及模板缓冲区中。前面曾提到，只有通过模板测试的像素才能渲染至后台缓冲区。对此，我们便将其设置为：仅当模板缓冲区中的值为1时，才能通过模板测试。这可以通过令`StencilRef`为1，且模板运算符为`D3D12_COMPARISON_FUNC_EQUAL`来实现。如此一来，只有模板缓冲区中元素数值为1的骷髅头镜像部分才能得以渲染。由于只有镜面可见部分所对应的模板缓冲区中元素数值为1，所以仅有这一范围内的骷髅头镜像才能被渲染出来。

5．最后，我们像往常那样将镜面渲染到后台缓冲区中。但是，为了能“透过”镜面观察骷髅头的镜像（它实际位于镜子的背面。虽说展现的是镜面内的镜像，但实际上是镜面背后的反射实物与镜面透明混合所得到的效果），我们就需要运用透明混合技术来渲染镜面。若非如此，则由于骷髅头镜像的深度值小于镜面的深度值，理所当然地会致使骷髅头镜像被镜子挡住。为此，我们只需为镜面定义一个新的材质配置实例：将其漫反射alpha通道分量设为0.3，使镜子的不透明度达到30%，并按第10.5.4小节中所述的透明混合状态来渲染镜面。

```
auto icemirror = std::make_unique<Material>();
icemirror->Name = "icemirror";
icemirror->MatCBIndex = 2;
```

```
    icemirror->DiffuseSrvHeapIndex = 2;
    icemirror->DiffuseAlbedo = xmFLOAT(1.0f,1.0f,1.0f,0.3f);
    icemirror->FresnelR0 = XMFLOAT3(0.1f, 0.1f, 0.1f);
    icemirror->Roughness = 0.5f;
```

上述设置可用下列混合公式来表示：

$$\boldsymbol{C} = 0.3 \cdot \boldsymbol{C}_{\text{src}} + 0.7 \cdot \boldsymbol{C}_{\text{dst}}$$

假设已经将骷髅头镜像的像素置于后台缓冲区内，那么，此时我们所看到的镜像颜色30%来自镜子（源像素），70%出自骷髅头镜像（目标像素）。

11.4.2　定义镜像的深度/模板状态

为了实现上述算法，我们要用到两个PSO对象。第一个用于在绘制镜面时标记模板缓冲区内镜面部分的像素，第二个则用于绘制镜面可见部分（即不被前侧实物所遮挡部分）内的骷髅头镜像。

```
//
// 用于标记模板缓冲区中镜面部分的PSO
//

// 禁止对渲染目标的写操作
CD3DX12_BLEND_DESC mirrorBlendState(D3D12_DEFAULT);
MirrorBlendState.RenderTarget[0].RenderTargetWriteMask =0;

D3D12_DEPTH_STENCIL_DESC mirrorDSS;
mirrorDSS.DepthEnable = true;
mirrorDSS.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ZERO;
mirrorDSS.DepthFunc = D3D12_COMPARISON_FUNC_LESS;
mirrorDSS.StencilEnable = true;
mirrorDSS.StencilReadMask = 0xff;
mirrorDSS.StencilWriteMask = 0xff;

mirrorDSS.FrontFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;
mirrorDSS.FrontFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;
```

```
mirrorDSS.FrontFace.StencilPassOp = D3D12_STENCIL_OP_REPLACE;
mirrorDSS.FrontFace.StencilFunc = D3D12_COMPARISON_FUNC_ALWAYS;

// 我们不渲染背面朝向的多边形，因而对这些参数的设置并不关心
mirrorDSS.BackFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;
mirrorDSS.BackFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;
mirrorDSS.BackFace.StencilPassOp = D3D12_STENCIL_OP_REPLACE;
mirrorDSS.BackFace.StencilFunc = D3D12_COMPARISON_FUNC_ALWAYS;

D3D12_GRAPHICS_PIPELINE_STATE_DESC markMirrorsPsoDesc = opaquePsoDesc;
markMirrorsPsoDesc.BlendState = mirrorBlendState;
markMirrorsPsoDesc.DepthStencilState = mirrorDSS;
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(
  &markMirrorsPsoDesc,
  IID_PPV_ARGS(&mPSOs["markStencilMirrors"])));

//
// 用于渲染模板缓冲区中反射镜像的PSO
//

D3D12_DEPTH_STENCIL_DESC reflectionsDSS;
reflectionsDSS.DepthEnable = true;
reflectionsDSS.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ALL;
reflectionsDSS.DepthFunc = D3D12_COMPARISON_FUNC_LESS;
reflectionsDSS.StencilEnable = true;
reflectionsDSS.StencilReadMask = 0xff;
reflectionsDSS.StencilWriteMask = 0xff;

reflectionsDSS.FrontFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;
reflectionsDSS.FrontFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;
reflectionsDSS.FrontFace.StencilPassOp = D3D12_STENCIL_OP_KEEP;
reflectionsDSS.FrontFace.StencilFunc = D3D12_COMPARISON_FUNC_EQUAL;

// 我们这里不对背面朝向的多边形进行渲染，因而这些配置是无足轻重的
reflectionsDSS.BackFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;
reflectionsDSS.BackFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;
reflectionsDSS.BackFace.StencilPassOp = D3D12_STENCIL_OP_KEEP;
reflectionsDSS.BackFace.StencilFunc = D3D12_COMPARISON_FUNC_EQUAL;

D3D12_GRAPHICS_PIPELINE_STATE_DESC drawReflectionsPsoDesc = opaquePsoDesc;
drawReflectionsPsoDesc.DepthStencilState = reflectionsDSS;
drawReflectionsPsoDesc.RasterizerState.CullMode = D3D12_CULL_MODE_BACK;
drawReflectionsPsoDesc.RasterizerState.FrontCounterClockwise = true;
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(
  &drawReflectionsPsoDesc,
  IID_PPV_ARGS(&mPSOs["drawStencilReflections"])));
```

11.4.3 绘制场景

以下代码概述了场景的绘制流程。为了清晰和简洁起见，这里省略了诸如“设置常量缓冲区数据”等的细枝末节（具体细节可参见例程的完整代码）。

```
// 绘制不透明的物体——地板、墙壁、骷髅头
auto passCB = mCurrFrameResource->PassCB->Resource();
mCommandList->SetGraphicsRootConstantBufferView(2,
  passCB->GetGPUVirtualAddress());
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)
    RenderLayer::Opaque]);

// 将模板缓冲区中可见的镜面像素标记为1
mCommandList->OMSetStencilRef(1);
mCommandList->SetPipelineState(mPSOs["markStencilMirrors"].Get());
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Mirrors]
);

// 只绘制镜子范围内的镜像（即仅绘制模板缓冲区中标记为1的像素）
// 注意，我们必须使用两个单独的渲染过程常量缓冲区（per-pass constant buffer）来
完成此工作，
// 一个存储物体镜像，另一个保存光照镜像
mCommandList->SetGraphicsRootConstantBufferView(2,
  passCB->GetGPUVirtualAddress() + 1 * passCBByteSize);
mCommandList->SetPipelineState(mPSOs["drawStencilReflections"].Get());
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Reflecte
d]);

// 恢复主渲染过程常量数据以及模板参考值
mCommandList->SetGraphicsRootConstantBufferView(2,
  passCB->GetGPUVirtualAddress());
mCommandList->OMSetStencilRef(0);

// 绘制透明的镜面，使镜像可以与之混合
mCommandList->SetPipelineState(mPSOs["transparent"].Get());
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Transpar
ent]);
```

关于以上代码还有一点需要注意，即在绘

制RenderLayer::Reflected层的时候如何来修改其渲染过程常量缓冲区。这是因为在绘制物体镜像的同时，还涉及场景中光照的镜像（即，物体的镜像也要有与之对应的光照）。光源本存于渲染过程常量缓冲区中，因此我们可以再额外创建一个渲染过程常量缓冲区，用以存储场景中光照的镜像。该常量缓冲区的设置方法如下：

```
PassConstants StencilApp::mMainPassCB;
PassConstants StencilApp::mReflectedPassCB;
void StencilApp::UpdateReflectedPassCB(const GameTimer& gt)
{
  mReflectedPassCB = mMainPassCB;

  XMVECTOR mirrorPlane = XMVectorSet(0.0f, 0.0f, 1.0f, 0.0f); // xy 平面
  XMMATRIX R = XMMatrixReflect(mirrorPlane);

  // 光照镜像
  for(int i = 0; i < 3; ++i)
  {
    XMVECTOR lightDir = XMLoadFloat3(&mMainPassCB.Lights[i].Direction);
    XMVECTOR reflectedLightDir = XMVector3TransformNormal(lightDir, R);
    XMStoreFloat3(&mReflectedPassCB.Lights[i].Direction, reflectedLightDir
);
  }

  // 将光照镜像的渲染过程常量数据存于渲染过程常量缓冲区中索引1的位置
  auto currPassCB = mCurrFrameResource->PassCB.get();
  currPassCB->CopyData(1, mReflectedPassCB);
}
```

11.4.4 绕序与镜像

当一个三角形被反射到某个平面上时（也就是此三角形在这一平面上的镜像），其绕序（winding order）并不会发生改变，正因如此，其平面法线的方向同样保持不变。所以，实际物体的外向法线在镜像中则变为了内向法线（见图11.5）。此时，为了纠正这一点，我们会告知

Direct3D将逆时针绕序的三角形看作是正面朝向，而将顺时针绕序的三角形看作背面朝向（这与我们之前的习惯刚好相反——见5.10.2节）。这实际上是对法线的方向也进行了“反射”，以此使镜像成为外向朝向。我们可以通过设置下列PSO光栅化属性来改变绕序的约定：

```
drawReflectionsPsoDesc.RasterizerState.FrontCounterClockwise = true;
```

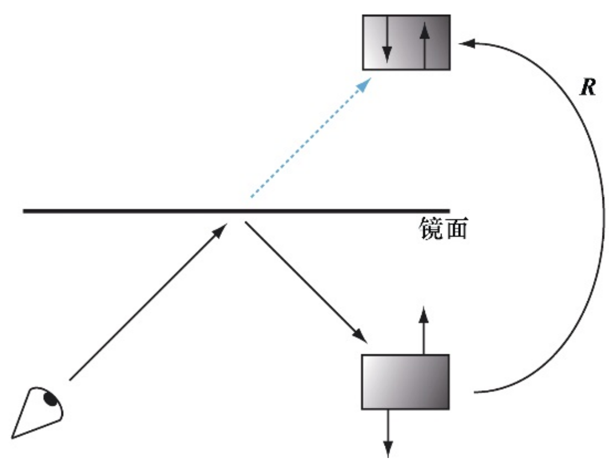


图11.5　多边形的法线不会随着反射操作而调转过来，这使得镜像的法线都变为内向朝向

11.5　实现平面阴影

注意

本节中的部分内容选自Frank D. Luna的*Introduction to 3D Game Programming with DirectX 9.0c: A Shader Approach*, 2006。出版社为Jones and Bartlett Learning, Burlington, MA. *www.jblearning.com*。现经许可摘录于此。

阴影有助于体现场景中的光源位置，用以加强场景的真实感。本节将展示如何实现平面阴影（planar shadow），即投射于平面内的阴影（见图11.6）。

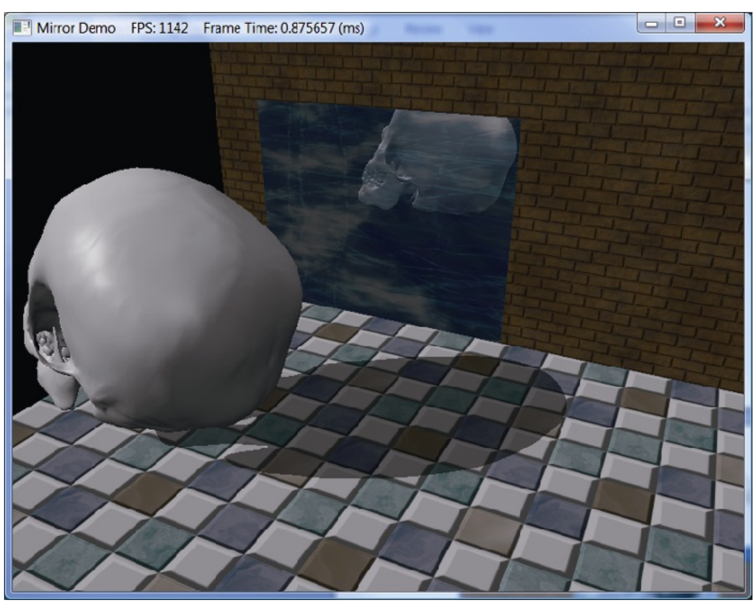


图11.6 “Stencil Demo”（模板）例程中主光源投射出的平面阴影

我们必须借助几何建模的方式首先找到物体经光照投向平面的阴影，从而渲染出平面阴影效果。这可以通过一些3D数学知识来轻松实现。接下来，再运用表示阴影的50%透明度黑色材质来渲染阴影区域中的三角形即可。渲染阴影这类工作有可能会引入名为“双重混合（也作双倍混合）”的渲染问题，这一概念会在后续小节中讲解，届时，我们将利用模板缓冲区来避免双重混合的发生。

11.5.1 平行光阴影

图11.7展示了由平行光源经物体所投射出的阴影。给定方向为$\boldsymbol{L}$的平行光源，并用$\boldsymbol{r}(t)=\boldsymbol{p}+t\boldsymbol{L}$来表示途经顶点$\boldsymbol{p}$的光线。光线$\boldsymbol{r}(t)$与阴影平面$(\boldsymbol{n},d)$的交点为$\boldsymbol{s}$。（读者可以从附录C中获取更多关于光线（射线）与平面相交的数学知识。）以此光源射出的光线照射到物体的各个顶点，用这些映射到平面上的交点集合便可以定义几何体所投射出的阴影形状。对于顶点$\boldsymbol{p}$来说，它的阴影投影可由下列公式求出：

$$\boldsymbol{s}=\boldsymbol{r}(t_s)=\boldsymbol{p}-\frac{\boldsymbol{n}\cdot\boldsymbol{p}+d}{\boldsymbol{n}\cdot\boldsymbol{L}}\boldsymbol{L} \qquad (11.1)$$

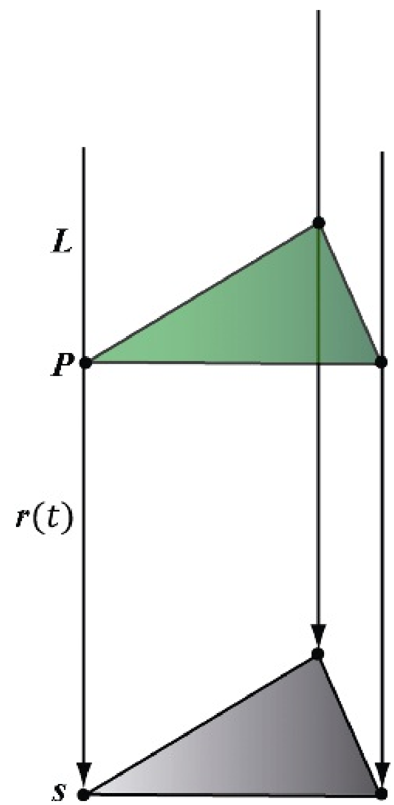


图11.7　平行光源及其投射阴影的示意图

光线与平面的相交测试细节可参见附录C。

式（11.1）可写作矩阵的形式。

$$\boldsymbol{s}' = [p_x \quad p_y \quad p_z \quad 1]\begin{bmatrix} \boldsymbol{n}\cdot\boldsymbol{L} - L_x n_x & -L_y n_x & -L_z n_x & 0 \\ -L_x n_y & \boldsymbol{n}\cdot\boldsymbol{L} - L_y n_y & -L_z n_y & 0 \\ -L_x n_z & -L_y n_z & \boldsymbol{n}\cdot\boldsymbol{L} - L_z n_z & 0 \\ -L_x d & -L_y d & -L_z d & \boldsymbol{n}\cdot\boldsymbol{L} \end{bmatrix}$$

我们称以上的4×4矩阵为方向光阴影矩阵（directional shadow matrix，也译作平行光阴影矩阵），用$\boldsymbol{S}_{dir}$表示。为了证明此矩阵与式（11.1）是等价的，我们用乘法运算来加以验证。首先可以看出，此矩阵改变了w分量，即$s_w = \boldsymbol{n}\cdot\boldsymbol{L}$。这样一来，当执行透视除法（参见第

5.6.3.4小节）时，$\boldsymbol{s}$中的每个坐标都会除以$\boldsymbol{n} \cdot \boldsymbol{L}$，这便是矩阵能做到式（11.1）中相应除法运算的原因。现在通过矩阵乘法来求出第i个投影顶点的坐标$s_i^{'}$，其中$i \in \{1, 2, 3$。在透视除法完成之后，我们有：

$$\begin{aligned} s_i^{'} &= \frac{(\boldsymbol{n} \cdot \boldsymbol{L})p_i - L_i n_x p_x - L_i n_y p_y - L_i n_z p_z - L_i d}{\boldsymbol{n} \cdot \boldsymbol{L}} \\ &= \frac{(\boldsymbol{n} \cdot \boldsymbol{L})p_i - (\boldsymbol{n} \cdot \boldsymbol{p} + d)L_i}{\boldsymbol{n} \cdot \boldsymbol{L}} \\ &= p_i - \frac{\boldsymbol{n} \cdot \boldsymbol{p} + d}{\boldsymbol{n} \cdot \boldsymbol{L}} L_i \end{aligned}$$

此结果与用式（11.1）求出的$\boldsymbol{s}$中的第i个坐标完全相同，因此$\boldsymbol{s} = \boldsymbol{s}'$。

为了运用阴影矩阵，我们将它与世界矩阵组合在一起。但是，在世界变换之后，由于透视除法还没有执行，因而几何体的阴影还未被投射到阴影平面上。此时便出现了一个问题：若$s_w = \boldsymbol{n} \cdot \boldsymbol{L} < 0$，则$w$坐标将变为负值。在透视投影的处理过程中，我们一般将z坐标复制到w坐标，若w坐标为负值则表明此点位于视锥体之外而应将其裁剪掉（裁剪操作在透视除法之前的齐次空间内执行）。这对于平面阴影来讲是个大问题，因为除了计算透视除法之外，我们还要用w坐标来实现阴影效果。图11.8就展示了这样一种$\boldsymbol{n} \cdot \boldsymbol{L} < 0$却存在阴影的情况，但此时这个阴影却无法显示出来。

为了纠正这个问题，我们用指向无穷远处光源的方向向量$\tilde{\boldsymbol{L}} = -\boldsymbol{L}$来取代光线方向向量$\boldsymbol{L}$。可以看出，$\boldsymbol{r}(t) = \boldsymbol{p} + t\boldsymbol{L}$与$\boldsymbol{r}(t) = \boldsymbol{p} + t\tilde{\boldsymbol{L}}$定义的是相同的3D直线，且该直线与平面之间的交点也是一致的（利用不同的交点参数值t_s来弥补$\tilde{\boldsymbol{L}}$与$\boldsymbol{L}$之间的符号差异）。因此使用$\tilde{\boldsymbol{L}} = -\boldsymbol{L}$会得到

与$\boldsymbol{L}$相同的计算结果，但是前者会保证$\boldsymbol{n}\cdot\boldsymbol{L}>0$，以此来绕开$w$坐标为负值的这个坑。

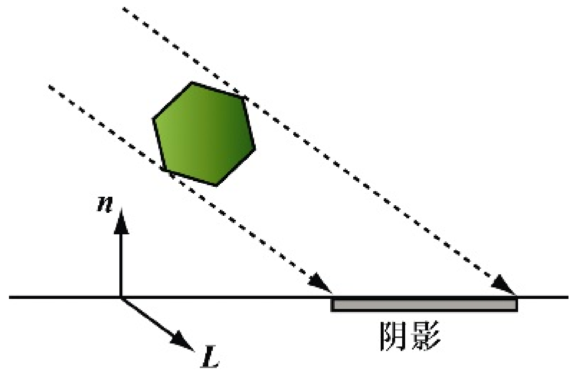


图11.8　图中所示的是一种$\boldsymbol{n}\cdot\boldsymbol{L}<0$的情况

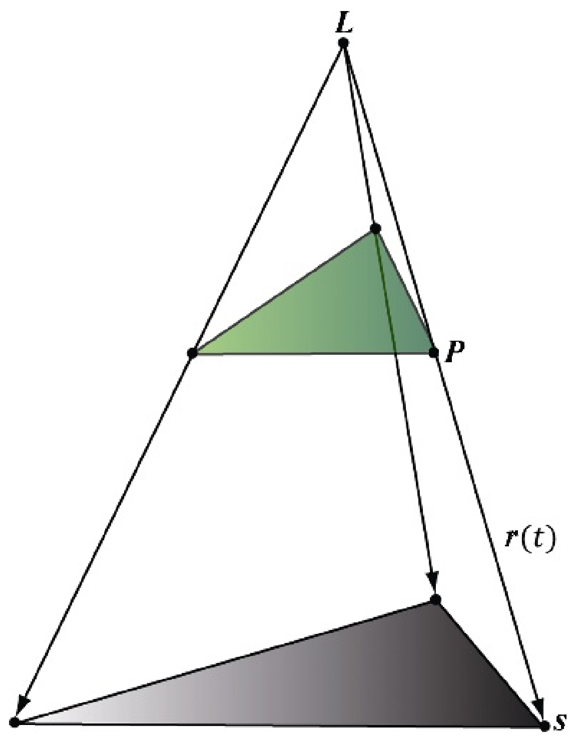


图11.9　点光源及其投射阴影的示意图

11.5.2　点光阴影

图11.9展示了位于点$\boldsymbol{L}$处的点光源所投射出的物体阴影。从点光源发出的途经任意顶点$\boldsymbol{p}$的光线可由$\boldsymbol{r}(t)=\boldsymbol{p}+t(\boldsymbol{p}-\boldsymbol{L})$来表示。光线$\boldsymbol{r}(t)$与阴影平面$(\boldsymbol{n},d)$的交点为$\boldsymbol{s}$。以此光源发出的光线经过物体的每个顶点，这映射在平面的交点集合便定义了几何体所投射出的阴影形状。对于顶点$\boldsymbol{p}$而言，其阴影投影可以表示为：

$$\boldsymbol{s}=\boldsymbol{r}(t_s)=\boldsymbol{p}-\frac{\boldsymbol{n}\cdot\boldsymbol{p}+d}{\boldsymbol{n}\cdot(\boldsymbol{p}-\boldsymbol{L})}(\boldsymbol{p}-\boldsymbol{L}) \tag{11.2}$$

式（11.2）也可以写作矩阵方程：

$$\boldsymbol{S}_{\text{point}}=\begin{bmatrix} \boldsymbol{n}\cdot\boldsymbol{L}+d-L_x n_x & -L_y n_x & -L_z n_x & -n_x \\ -L_x n_y & \boldsymbol{n}\cdot\boldsymbol{L}+d-L_y n_y & -L_z n_y & -n_y \\ -L_x n_z & -L_y n_z & \boldsymbol{n}\cdot\boldsymbol{L}+d-L_z n_z & -n_z \\ -L_x d & -L_y d & -L_z d & \boldsymbol{n}\cdot\boldsymbol{L} \end{bmatrix}$$

为了证明此矩阵等价于式（11.2），我们就以上一节中同样的办法，用矩阵进行乘法运算。观察到最后一列中并没有0项，则有：

$$\begin{aligned} s_w &= -p_x n_x - p_y n_y - p_z n_z + \boldsymbol{n}\cdot\boldsymbol{L} \\ &= -\boldsymbol{p}\cdot\boldsymbol{n}+\boldsymbol{n}\cdot\boldsymbol{L} \\ &= -\boldsymbol{n}\cdot(\boldsymbol{p}-\boldsymbol{L}) \end{aligned}$$

这就是式（11.2）中分母部分的相反数，我们可以通过将分子与分母同时乘以−1，使二者一致。

注意 Note

对于点光与平行光而言，$\boldsymbol{L}$充当着不同的角色。在使用点光时，$\boldsymbol{L}$

定义了点光源的位置。在使用平行光时，我们却用$\boldsymbol{L}$来定义指向无穷远处光源的方向向量（即与平行光光线传播方向相反的向量）。

11.5.3 通用阴影矩阵

我们可通过齐次坐标创建出一个能同时应用于点光与方向光的通用阴影矩阵。

1．如果$L_{\text{w}}=0$，则$\boldsymbol{L}$表示指向无穷远处光源的方向向量（即与平行光光线传播方向相反的向量）。

2．如果$L_{\text{w}}=1$，则$\boldsymbol{L}$表示点光的位置。

接下来，我们用下列阴影矩阵（shadow matrix）来表示由顶点$\boldsymbol{p}$到其投影$\boldsymbol{s}$的变换：

$$\boldsymbol{S}=\begin{bmatrix} \boldsymbol{n}\cdot\boldsymbol{L}+dL_w-L_xn_x & -L_yn_x & -L_zn_x & -L_wn_x \\ -L_xn_y & \boldsymbol{n}\cdot\boldsymbol{L}+dL_w-L_yn_y & -L_zn_y & -L_wn_y \\ -L_xn_z & -L_yn_z & \boldsymbol{n}\cdot\boldsymbol{L}+dL_w-L_zn_z & -L_wn_z \\ -L_xd & -L_yd & -L_zd & \boldsymbol{n}\cdot\boldsymbol{L} \end{bmatrix}$$

显然，如果$L_{\text{w}}=0$，则$\boldsymbol{S}$将化简为$\boldsymbol{S}_{\text{dir}}$，若$L_{\text{w}}=1$，则$\boldsymbol{S}$将化简为$\boldsymbol{S}_{\text{point}}$。

DirectX的数学库提供了以下函数，用以构建在特定平面内投射阴影所用的相应阴影矩阵，若$w=0$表示平行光，而$w=1$则表示点光：

```
inline XMMATRIX XM_CALLCONV XMMatrixShadow(
  FXMVECTOR ShadowPlane,
  FXMVECTOR LightPosition);
```

[Blinn96]与[Möller02]可作为拓展阅读，它们都讨论了平面阴影的相关内容。

11.5.4 使用模板缓冲区防止双重混合

将物体的几何形状投射到平面而形成阴影时，可能（实际上也经常出现）会有两个甚至更多的平面阴影三角形相互重叠。若此时用透明度这一混合技术来渲染阴影，则这些三角形的重叠部分会混合多次，使之看起来更暗。图11.10即说明了这一点。

这个问题可以通过模板缓冲区来解决。

1．首先，保证参与渲染阴影的模板缓冲区中的阴影范围像素都已被清理为0。由于“Stencil Demo”演示程序只向地面投射一种阴影，所以这样做也是合情合理的，而且我们仅需修改阴影的模板缓冲区中的像素即可。

2．设置模板测试，使之仅接受模板缓冲区中元素为0的像素。如果通过模板测试，则将相应模板缓冲区值增为1。

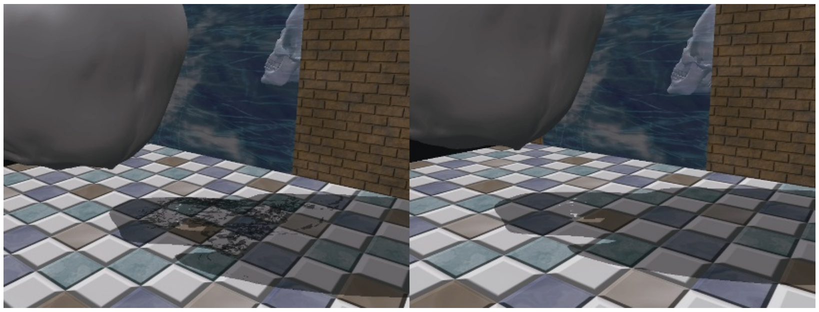


图11.10　可以看到，左图阴影中存在颜色更暗的“粉刺”（acne）部分。这些区域即为骷髅头平面投影三角形重合的部分，由此便导致了“双重混合”的发生。因为右图没有进行双重混合，所以渲染出的是正确的阴影效果

在第一次渲染阴影像素时，由于模板缓冲区元素为0，因而模板测试会成功。渲染该像素的同时，我们也会将对应的模板缓冲区元素增加为1。这样一来，如果试图覆写已被渲染过的区域，则模板测试会失败。这将防止同一像素被绘制多次，继而阻止双重混合的发生。

11.5.5　编写阴影部分的代码

我们把用于绘制阴影的材质定义为具有50%透明度的黑色材质：

```
auto shadowMat = std::make_unique<Material>();
shadowMat->Name = "shadowMat";
shadowMat->MatCBIndex = 4;
shadowMat->DiffuseSrvHeapIndex = 3;
shadowMat->DiffuseAlbedo = XMFLOAT4(0.0f, 0.0f, 0.0f, 0.5f);
shadowMat->FresnelR0 = XMFLOAT3(0.001f, 0.001f, 0.001f);
shadowMat->Roughness = 0.0f;
```

为了防止双重混合，我们用下列的深度/模板状态来设置PSO：

```
// 以下列深度/模板状态来防止双重混合的发生
D3D12_DEPTH_STENCIL_DESC shadowDSS;
shadowDSS.DepthEnable = true;
```

```
shadowDSS.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ALL;
shadowDSS.DepthFunc = D3D12_COMPARISON_FUNC_LESS;
shadowDSS.StencilEnable = true;
shadowDSS.StencilReadMask = 0xff;
shadowDSS.StencilWriteMask = 0xff;

shadowDSS.FrontFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;
shadowDSS.FrontFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;

// 由于并不渲染背面朝向的多边形，因此这些配置都是无关紧要的
shadowDSS.BackFace.StencilFailOp = D3D12_STENCIL_OP_KEEP;
shadowDSS.BackFace.StencilDepthFailOp = D3D12_STENCIL_OP_KEEP;
shadowDSS.BackFace.StencilPassOp = D3D12_STENCIL_OP_INCR;
shadowDSS.BackFace.StencilFunc = D3D12_COMPARISON_FUNC_EQUAL;

D3D12_GRAPHICS_PIPELINE_STATE_DESC shadowPsoDesc = transparentPsoDesc;
shadowPsoDesc.DepthStencilState = shadowDSS;
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(
  &shadowPsoDesc,
  IID_PPV_ARGS(&mPSOs["shadow"])));
```

接着用`StencilRef`值为0的阴影PSO来绘制骷髅头阴影：

```
// 绘制阴影
mCommandList->OMSetStencilRef(0);
mCommandList->SetPipelineState(mPSOs["shadow"].Get());
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Shadow]);
```

在这里，骷髅头阴影渲染项的世界矩阵是这样计算的：

```
// 更新阴影的世界矩阵
XMVECTOR shadowPlane = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f); // xz 平面
XMVECTOR toMainLight = -XMLoadFloat3(&mMainPassCB.Lights[0].Direction);
XMMATRIX S = XMMatrixShadow(shadowPlane, toMainLight);
XMMATRIX shadowOffsetY = XMMatrixTranslation(0.0f, 0.001f, 0.0f);
XMStoreFloat4x4(&mShadowedSkullRitem->World, skullWorld * S * shadowOffset
Y);
```

注意，我们将投影网格沿着y轴做了少量的偏移调整，以防发生深度冲突（z-fighting），所以阴影网格不会与地板网格相交，得到的最终

效果是阴影会略高于地板。如果这两种网格相交，则由于深度缓冲区的精度限制，将导致地板与阴影的网格像素为了各自的完全显现而发生闪烁的现象[2]。

11.6 小结

1．模板缓冲区是一种离屏缓冲区，我们可以通过它来阻止特定像素片段向后台缓冲区的渲染操作。由于模板缓冲区与深度缓冲区分辨率相同，所以两者可以联合使用。深度/模板缓冲区的有效格式为DXGI_FORMAT_D32_FLOAT_S8X24_UINT与DXGI_FORMAT_D24_UNORM_S8_UINT。

2．是否可向特定的像素执行写操作都取决于模板测试，该测试过程如下：

```
if( StencilRef & StencilReadMask ⊴ Value & StencilReadMask )
```

```
    accept pixel
  else
    reject pixel
```

其中，运算符⊴是枚举类型D3D12_COMPARISON_FUNC中定义的函数之一。StencilRef、 StencilReadMask与比较运算符⊴皆是程序中定义且以Direct3D深度/模板API设置的比较数值。Value即当前正在比较中的模板缓冲区像素值。

3．深度/模板状态是PSO描述的一部分。通过填写D3D12_GRAPHICS_PIPELINE_STATE_DESC::DepthStencilState字段便可配置深度/模板状态，而DepthStencilState的类型则为D3D12_DEPTH_STENCIL_DESC。

4. 模板参考值

由ID3D12GraphicsCommandList::OMSetStencilRef方法来设置，需要传入一个无符号整数作为参数来指定模板参考值。

11.7　练习

1．证明：如果$L_w = 0$，则通用阴影矩阵$\boldsymbol{S}$将化简为$\boldsymbol{S}_{\text{dir}}$；而$L_w = 1$，则$\boldsymbol{S}$将化简为$\boldsymbol{S}_{\text{point}}$。

2．就像我们在11.5.1节中计算方向光平面阴影所做的那样，通过对顶点的每个分量进行矩阵乘法来证明$\boldsymbol{s} = \boldsymbol{p} - \dfrac{\boldsymbol{n} \cdot \boldsymbol{p} + d}{\boldsymbol{n} \cdot (\boldsymbol{p} - \boldsymbol{L})}(\boldsymbol{p} - \boldsymbol{L}) = \boldsymbol{p}\boldsymbol{S}_{\text{point}}$。

3．修改“Stencil Demo”演示程序，生成图11.1左图所示的效果。

4．修改“Stencil Demo”演示程序，生成图11.10左图所示的效果。

5．通过下列步骤修改“Stencil Demo”演示程序。首先用下面的深度状态来绘制一堵墙：

```
depthStencilDesc.DepthEnable   = false;
depthStencilDesc.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ALL;
depthStencilDesc.DepthFunc     = D3D12_COMPARISON_FUNC_LESS;
```

接下来，再以如下深度状态来绘制墙后的骷髅头：

```
depthStencilDesc.DepthEnable   = true;
depthStencilDesc.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ALL;
depthStencilDesc.DepthFunc     = D3D12_COMPARISON_FUNC_LESS;
```

墙壁会遮挡骷髅头吗？请做出解释。如果按下列设置来绘制墙壁会发生什么呢？

```
depthStencilDesc.DepthEnable  = true;
depthStencilDesc.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ALL;
depthStencilDesc.DepthFunc    = D3D12_COMPARISON_FUNC_LESS;
```

注意，此练习不涉及模板缓冲区，所以应将它禁用。

6．修改“Stencil Demo”演示程序。若以不调转三角形的绕序约定为前提，能否正确地渲染出骷髅头镜像？

7．修改第10章中的“Blend Demo”（混合）演示程序：在场景的中心绘制一个（没有上底与下底的）圆柱体。该圆柱体的纹理采用的是60帧螺旋电流动画，在本章目录下[3]找到此素材，并以累加混合（additive blending）的方式来实现此效果。图11.11展示了此示例的最终效果。

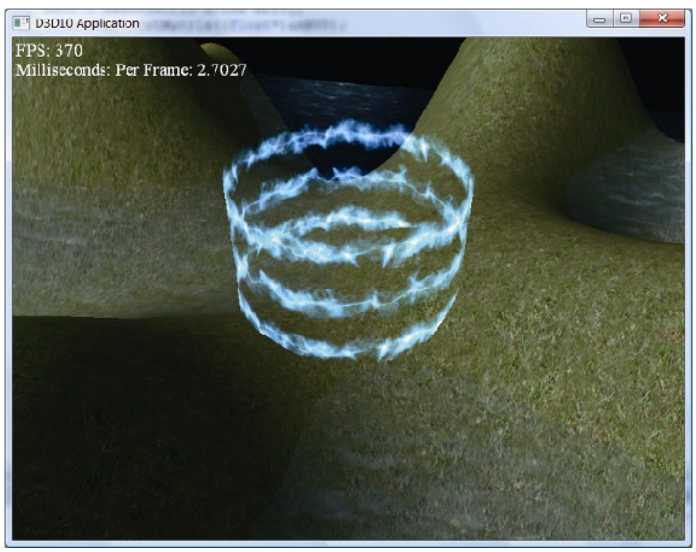


图11.11　练习7答案的效果

提示

可参考10.5.5节中，以累加混合的方式渲染几何体时所使用的深度状态。

8．深度复杂性（depth complexity，或深度复杂度）是指通过深度测试竞争，向后台缓冲区中某一特定元素写入像素片段的次数。例如，一个本已绘制好的像素可能被更靠近摄像机的像素所覆写（整个场景虽然绘制完成，但在确定最靠近摄像机的像素之前，这种重复绘制的情况还是会发生若干次）。如图11.12所示，像素$\boldsymbol{p}$的深度复杂性为3，因为在该像素上共有3种像素片段。

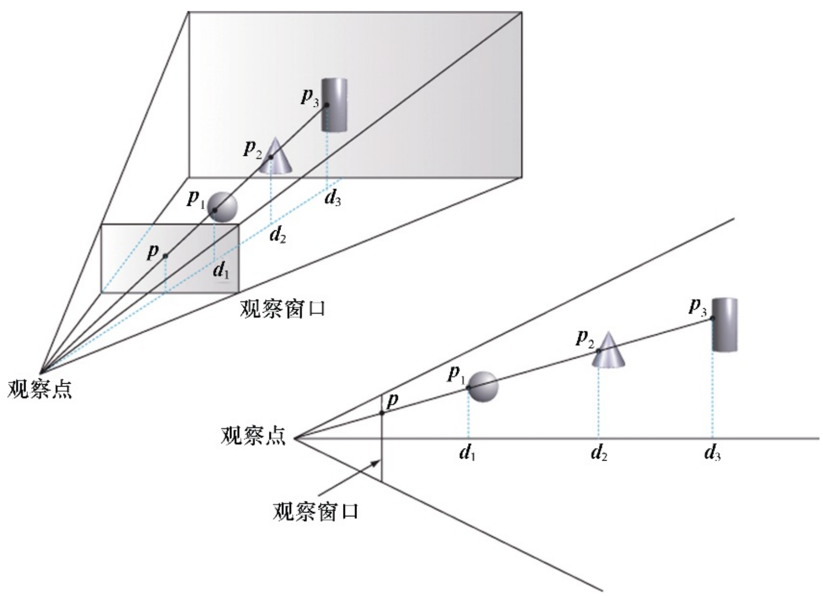


图11.12　竞相渲染到投射窗口上某一像素的多个像素片段。在此场景中，像素$\boldsymbol{p}$的深度复杂性为3

事实上，显卡可能会在每一帧中将某一像素填充多次。这种重复绘制（overdraw）的行为会对性能造成影响，因为显卡把时间浪费在了覆

写最终看不到的像素上。因此，度量场景中的深度复杂性对于性能分析是很有意义的。

我们可按下列方法来衡量深度复杂性：用模板缓冲区来渲染场景并把它作为计数器，即将模板缓冲区中每个像素的初始值清理为0，在每次处理像素片段时，用D3D12_STENCIL_OP_INCR方法令其计数递增。对于任一像素片段的写入操作而言，应当总是对它相应的模板缓冲区元素值加1，所以我们采用模板比较函数D3D12_COMPARISON_FUNC_ALWAYS。举个例子，在一帧绘制完毕后，若第i行、第j列像素的对应模板缓冲区元素值为5，则表示该像素在此帧的绘制过程中共被写入了5次像素片段（即此像素的深度复杂性为5）。注意，严格来讲，在统计深度复杂性时，我们只需将场景渲染到模板缓冲区即可。

为了使（存于模板缓冲区中的）深度复杂性可视化，可进行如下处理。

a. 令颜色c_k与深度复杂性k关联起来。比如说，蓝色对应深度复杂性1、绿色对应深度复杂性2、红色对应深度复杂性3等。（在极其复杂的场景中，一些像素的深度复杂性值可能非常大，因此，我们也许不希望为每个级别都逐一设定颜色。此时，我们便可以将一系列没有交集的连续级别关联为同一种颜色。例如，设置深度复杂性为1～5的像素为蓝色，深度复杂性为6～10的像素为绿色等。）

b. 设定模板缓冲区的运算方法为D3D12_STENCIL_OP_KEEP，即我

们不会对它进行任何修改。（当随着场景的渲染过程而统计深度复杂性时，我们用D3D12_STENCIL_OP_INCR来修改模板缓冲区。但是，为模板缓冲区的可视化而编写代码时，我们仅需从模板缓冲区读取数据而不应向其写入数据。）

c. 对于每种深度复杂性的级别k而言：

（i）设置模板比较函数为D3D12_COMPARISON_FUNC_EQUAL，且设定模板参考值为k。

（ii）用4种颜色c_k来绘制整个投影窗口内容。注意，根据前面所设置的模板比较函数与模板参考值可以判断，只有深度复杂性为k的像素才能被着色并显示出来。

经过这一系列的配置过程，我们就能根据每个像素自身的深度复杂性来为它上色，并以此来方便地观察场景中深度复杂性的分布情况。对于此练习来说，我们借用第10章中的“Blend Demo”演示程序来渲染出其场景的深度复杂性。图11.13展示了该示例的一个效果。

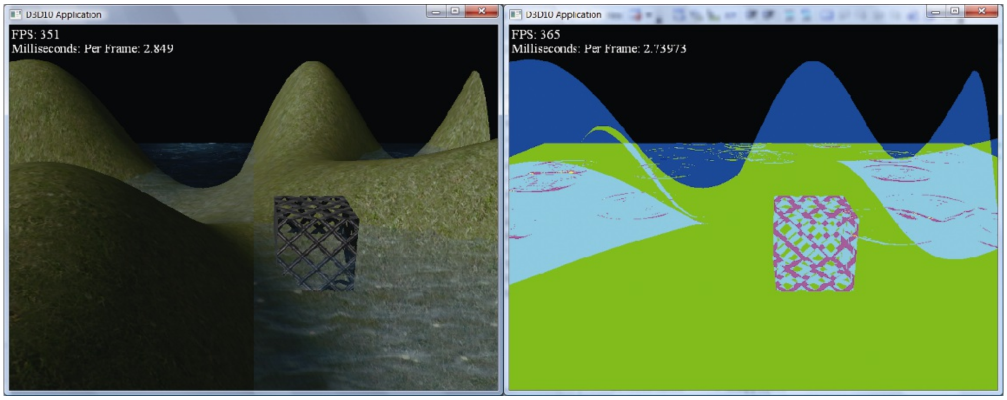


图11.13　练习8答案的对比效果

注意 Note

深度测试发生于渲染流水线像素着色器阶段之后的输出合并阶段。这意味着即使某像素片段最终可能被深度测试丢弃，它也要经像素着色器处理一番。然而，现代硬件在处理像素着色器之前会执行一种称为“提前z测试”（early z-test）的深度测试。如此一来，一个在后续处理过程中将被剔除的像素片段，可能在被开销很高的像素着色器处理之前便被舍去了。要使用这种优化技术，我们应当尝试按距离摄像机由近及远的顺序来渲染非混合的游戏对象。按照这种做法，离摄像机最近的物体将被首先绘制，其后的物体会在提前z测试中不予通过，继而不参与后续的处理流程。若处理的场景因较高的深度复杂性而需要大量的重新绘制工作，那么，应用提前z测试所得到的优化效果将是可想而知的。我们不能通过Direct3D API来控制提前z测试，而负责该功能的往往是图形驱动。例如，若像素着色器将修改像素片段的深度值，那么便不能进行提前z测试。这是因为倘若像素着色器修改了深度值，那么可能会对深度测试造成影响。所以，在这种情况下，像素着色器必须在深度测试之前执行。

注意 Note

前面提到，在像素着色器中能修改像素的深度值。但是该怎样来实现这种操作呢？正如我们目前所做的，像素着色器不仅能输出单个颜色向量，还能输出结构体：

```
struct PixelOut
{
  float4 color : SV_Target;
  float depth : SV_Depth;
};

PixelOut PS(VertexOut pin)
{
  PixelOut pout;

  // 常规的像素相关处理

  pout.Color = float4(litColor, alpha);

  // 把像素深度值设置在归一化范围[0, 1]内
  pout.depth = pin.PosH.z - 0.05f;

  return pout;
}
```

SV_Position元素的z坐标(pin.PosH.z)表示的是未加改动的像素深度值。通过使用特殊的系统值语义SV_Depth，便可以使像素着色器输出经过修改的深度值。

---

9. 另一种实现深度复杂性可视化的方式是使用加法混合技术。首先清理后台缓冲区，使之为黑色，并禁用深度测试。其次，设源混合因子与目标混合因子都为D3D12_BLEND_ONE，再将混合运算方法置为D3D12_BLEND_OP_ADD，使混合公式变为$C = C_{src} + C_{dst}$。观察此公

式可知，针对每个像素，我们将其所有像素片段的颜色都累加在一起并回写。最后，再用像素着色器以类似于(0.05, 0.05, 0.05)这种低强度的颜色输出方式来渲染场景中的所有物体。像素重新绘制的次数越多，这种低强度颜色也就累加得越多，对应像素就愈发明亮。例如，若某像素被重复绘制了10次，则该像素的颜色强度将变为(0.5, 0.5, 0.5)。这样一来，在场景渲染完成之后，我们通过观察像素的颜色强度便可了解场景中深度复杂性的分布情况。请用第10章中的“Blend Demo”演示程序作为测试场景，并以上述方式来测试其深度复杂性。

10．叙述如何来统计深度测试成功的像素数量。再阐明怎样才能计算出深度测试失败的像素数量。

11．修改“Stencil Demo”演示程序，把地板与骷髅头都映（反）射到镜子当中。

12．从渲染项阴影的世界矩阵中移除垂直方向上（*z*坐标）的偏移量，以便观察深度冲突。

---

[1] 在预览版中，枚举类型`D3D12_COMPARISON_FUNC`中成员的定义都形如`D3D12_COMPARISON_XOXO`。

[2] 两者在同一平面时，会相互打架（fighting），争夺在*z*轴上的高度，也就是显示权（高的就能抢镜露脸）。你死我活之际，显示权反复易主，从而互有显示的时段，造成闪烁的效果。

[3] 位于本书DirectX 11版第10章的BoltAnim文件夹中。

第12章　几何着色器

如果不启用曲面细分（tessellation）这一环节，那么几何着色器（geometry shader）这个可选阶段便会位于顶点着色器与像素着色器之间。顶点着色器以顶点作为输入数据，而几何着色器的输入数据则是完整的图元。例如，如果要绘制三角形列表（triangle list），则几何着色器程序实际将对列表中的每个三角形T执行下列操作：

```
for(UINT i = 0; i < numTriangles; ++i)
  OutputPrimitiveList = GeometryShader( T[i].vertexList );
```

可以注意到，几何着色器以每个三角形的3个顶点作为输入，且输出的是对应的图元列表。与顶点着色器不能销毁或创建顶点不同，几何着色器的亮点便是可以创建或销毁几何图形，此功能使GPU实现一些有趣的效果成为可能。比如说，借助几何着色器可以将输入的图元扩展为一个或更多其他类型的图元，或者能根据某些条件而选择不输出图元。注意，几何着色器的输出图元类型不一定与输入图元的类型相同。例如，几何着色器的一个常见拿手好戏即是将一个点扩展为一个四边形（即两个三角形）。

几何着色器所输出的图元由顶点列表定义而成。在退出几何着色器时，必将顶点的位置变换到齐次裁剪空间。换言之，经过几何着色器阶段的处理后，我们就得到了位于齐次裁剪空间中由一系列顶点所定义的多个图元。这些顶点会同样历经投影（齐次除法）与光栅化等后续步

骤。

学习目标：

1．学习如何编写几何着色器。

2．探究如何通过几何着色器来高效地实现公告牌技术（billboard）。

3．了解自动生成图元ID及其相关的应用。

4．研究如何创建和使用纹理数组，并认识到它们为何如此实用。

5．理解如何运用alpha-to-coverage技术来辅助解决alpha裁剪（alpha cutout）的失真问题。

12.1　编写几何着色器

几何着色器的编写方式比较接近于顶点着色器和像素着色器，当然也存在若干区别。下列代码展示了几何着色器的一般编写格式：

```
[maxvertexcount(N)]
void ShaderName (
 PrimitiveType InputVertexType InputName[NumElements],
 inout StreamOutputObject<OutputVertexType> OutputName)
{
    // 几何着色器的具体实现
}
```

我们必须先指定几何着色器单次调用所输出的顶点数量最大值（每个图元都会调用一次几何着色器，走一遍其中的处理流程）。对此，可以使用下列属性语法来设置着色器定义之前的最大顶点数量：

```
  [maxvertexcount(N)]
```

其中，N是几何着色器单次调用所输出的顶点数量最大值。几何着色器每次输出的顶点个数都可能各不相同，但是这个数量却不能超过之前定义的最大值。出于对性能方面的考量，我们应当令`maxvertexcount`的值尽可能地小。相关资料显示[NVIDIA08]，在GS（即几何着色器的缩写，geometry shader）每次输出的标量数量在1～20时，它将发挥出最佳的性能；而当GS每次输出的标量数量保持在27～40时，它的性能将下降到峰值性能的50%。每次调用几何着色器所输出的标量个数为：`maxvertexcount`与输出顶点类型结构体中标量个数的乘积[1]。在实践中完全满足这些限制是比较困难的，所以我们或取

比最佳性能稍差的解决方案，或干脆选择另一种与几何着色器无关的实现方法。当然，这里一定也要考虑到其他方法所带来的弊端——也许还不如直接用几何着色器来实现给力。再者，从2008年[NVIDIA08]（第一代几何着色器）至今，几何着色器的相关方面也有了不少的改良。

几何着色器有输入、输出共两个参数（实际上它可以拥有更多的参数，但这又是另一个主题了，具体内容参见12.2.4节）。输入参数必须是一个定义有特定图元的顶点数组——点应输入一个顶点、线条要输入两个顶点、三角形需输入3个顶点、线及其邻接图元为4个顶点、三角形及其邻接图元则为6个顶点。几何着色器的输入顶点类型即为顶点着色器输出的顶点类型（例如`VertexOut`）。输入参数一定要以图元类型作为前缀，用以描述输入到几何着色器的具体图元类型。该前缀可以是下列类型之一：

1. `point`：输入的图元为点。

2. `line`：输入的图元为线列表或线条带。

3. `triangle`：输入的图元为三角形列表或三角形带。

4. `lineadj`：输入的图元为线列表及其邻接图元，或线条带及其邻接图元。

5. `triangleadj`：输入的图元为三角形列表及其邻接图元，或三角形带及其邻接图元。

注意 Note

向几何着色器输入的数据必须是完整的图元（例如组成线条的两个顶点、构成三角形的3个顶点等）。因此，几何着色器并不会区分输入的图元究竟是列表结构（list）还是带状结构（strip）。举个例子，若绘制的图元实际上是三角形带，但几何着色器仍会把三角形带视作多个三角形并分别进行单独的处理，即将每个三角形的3个顶点作为其输入数据。绘制带状结构的过程中会产生额外的开销，因为多个图元所共用的顶点在几何着色器中会被处理多次。

输出参数一定要标有`inout`修饰符。另外，它必须是一种流类型（stream type。即某种类型的流输出对象）。流类型存有一系列顶点，它们定义了几何着色器输出的几何图形。几何着色器可以通过内置方法`Append`向输出流列表添加单个顶点：

```
void StreamOutputObject<OutputVertexType>::Append(OutputVertexType v);
```

流类型本质上是一种模板类型（template type），其模板参数用以指定输出顶点的具体类型（如`GeoOut`）。流类型有如下3种。

1. `PointStream<OutputVertexType>`：一系列顶点所定义的点列表。

2. LineStream<OutputVertexType>：一系列顶点所定义的线条带。

3. TriangleStream<OutputVertexType>：一系列顶点所定义的三角形带。

几何着色器输出的多个顶点会构成图元，图元的输出类型由流类型（即PointStream、LineStream与TriangleStream）来指定。对于线条与三角形来说，几何着色器输出的对应图元必定是线条带与三角形带。而线条列表与三角形列表可借助内置函数RestartStrip来实现：

```
void StreamOutputObject<OutputVertexType>::RestartStrip();
```

比如，如果希望输出三角形列表，则需要在每次向输出流追加3个顶点之后调用RestartStrip[2]。

以下是一些几何着色器签名的具体用例。

```
// 示例1：GS最多输出4个顶点。输入的图元一根是线条，输出的是一个三角形带
[maxvertexcount(4)]
void GS(line VertexOut gin[2],
    inout TriangleStream<GeoOut> triStream)
{
    // 几何着色器的具体实现......
}
//
// 示例2：GS最多输出32个顶点。输入的图元是一个三角形，输出的是一个三角形带
//
[maxvertexcount(32)]
void GS(triangle VertexOut gin[3],
    inout TriangleStream<GeoOut> triStream)
{
    // 几何着色器的具体实现......
}
//
```

```
// 示例3: GS至多输出4个顶点。输入的图元是一个点，输出的是一个三角形带
//
[maxvertexcount(4)]
void GS(point VertexOut gin[1],
    inout TriangleStream<GeoOut> triStream)
{
    // 几何着色器的主体...
}
```

下列几何着色器详细地展示了Append与RestartStrip方法的调用过程。此示例会将输入的三角形进行细分（见图12.1），并输出细分后的4个小三角形：

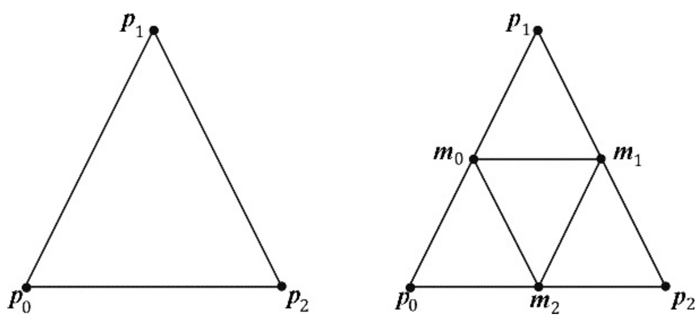


图12.1　将三角形细分为大小相等的4个小三角形。通过观察可以发现，新添加的3个顶点均位于原三角形边上的中点

```
struct VertexOut
{
    float3 PosL   : POSITION;
    float3 NormalL : NORMAL;
    float2 Tex    : TEXCOORD;
};

struct GeoOut
{
    float4 PosH   : SV_POSITION;
    float3 PosW   : POSITION;
    float3 NormalW : NORMAL;
    float2 Tex    : TEXCOORD;
    float FogLerp : FOG;
};

void Subdivide(VertexOut inVerts[3], out VertexOut outVerts[6])
```

```
{
    //       1
    //       *
    //      / \
    //     /   \
    //  m0*-----*m1
    //   / \   / \
    //  /   \ /   \
    // *-----*-----*
    // 0    m2     2

    VertexOut m[3];

    // 计算三角形边上的中点
    m[0].PosL = 0.5f*(inVerts[0].PosL+inVerts[1].PosL);
    m[1].PosL = 0.5f*(inVerts[1].PosL+inVerts[2].PosL);
    m[2].PosL = 0.5f*(inVerts[2].PosL+inVerts[0].PosL);

    // 把顶点投影到单位球面上
    m[0].PosL = normalize(m[0].PosL);
    m[1].PosL = normalize(m[1].PosL);
    m[2].PosL = normalize(m[2].PosL);

    // 求出法线
    m[0].NormalL = m[0].PosL;
    m[1].NormalL = m[1].PosL;
    m[2].NormalL = m[2].PosL;

    // 对纹理坐标进行插值
    m[0].Tex = 0.5f*(inVerts[0].Tex+inVerts[1].Tex);
    m[1].Tex = 0.5f*(inVerts[1].Tex+inVerts[2].Tex);
    m[2].Tex = 0.5f*(inVerts[2].Tex+inVerts[0].Tex);

    outVerts[0] = inVerts[0];
    outVerts[1] = m[0];
    outVerts[2] = m[2];
    outVerts[3] = m[1];
    outVerts[4] = inVerts[2];
    outVerts[5] = inVerts[1];
};

void OutputSubdivision(VertexOut v[6],
   inout TriangleStream<GeoOut> triStream)
{
    GeoOut gout[6];

    [unroll]
    for(int i = 0; i < 6; ++i)
```

```
    {
        // 将顶点变换到世界空间
        gout[i].PosW = mul(float4(v[i].PosL, 1.0f), gWorld).xyz;
        gout[i].NormalW = mul(v[i].NormalL,(float3x3)gWorldInvTranspose);

        // 把顶点变换到齐次裁剪空间
        gout[i].PosH = mul(float4(v[i].PosL, 1.0f), gWorldViewProj);
        gout[i].Tex  = v[i].Tex;
    }

    //       1
    //       *
    //      / \
    //     /   \
    //  m0*-----*m1
    //   / \   / \
    //  /   \ /   \
    // *-----*-----*
    // 0    m2     2

    // 我们可以将细分的小三角形绘制到两个三角形带中去:
    //    三角形带1: 底端的3个三角形
    //    三角形带2: 顶部的三角形

    [unroll]
    for(int j = 0; j < 5; ++j)
    {
        triStream.Append(gout[j]);
    }
    triStream.RestartStrip();

    triStream.Append(gout[1]);
    triStream.Append(gout[5]);
    triStream.Append(gout[3]);
}

[maxvertexcount(8)]
void GS(triangle VertexOut gin[3], inout TriangleStream<GeoOut>)
{
    VertexOut v[6];
    Subdivide(gin, v);
    OutputSubdivision(v, triStream);
}
```

几何着色器的编译过程也与顶点着色器和像素着色器如出一辙。如

果TreeSprite.hlsl文件中有名为GS的几何着色器，则可以用下列方法将其编译为字节码：

```
mShaders["treeSpriteGS"] = d3dUtil::CompileShader(
  L"Shaders\\TreeSprite.hlsl", nullptr, "GS", "gs_5_0");
```

就像顶点着色器与像素着色器一样，要将指定的几何着色器作为流水线状态对象（pipeline state object，PSO）的一部分，以此将它绑定到渲染流水线上：

```
D3D12_GRAPHICS_PIPELINE_STATE_DESC treeSpritePsoDesc = opaquePsoDesc;
...
treeSpritePsoDesc.GS =
{
    reinterpret_cast<BYTE*>(mShaders["treeSpriteGS"]->GetBufferPointer()),
    mShaders["treeSpriteGS"]->GetBufferSize()
};
```

注意 Note

若给出一个输入图元，几何着色器也可以根据某些条件而选择不输出任何数据。通过这种方式，几何着色便可以轻易地“销毁”几何图形，这对于一些算法的实现来讲是很有帮助的。

注意 Note

如果没有向几何着色器输入组装完整图元所需的足够顶点，将会导致部分图元的遗失[3]。

12.2　以公告牌技术实现森林效果

12.2.1　概述

当树与树之间的距离较远时，就轮到公告牌（billboard，也称公告板等）技术大显神威了，即以绘有3D树木图片的四边形来替代对整棵3D树的渲染（见图12.2）。从远处看去，公告牌技术往往并不会露出破绽。这里还有一个小秘诀，就是使公告牌总是面向摄像机（否则很容易露馅儿）[4]。

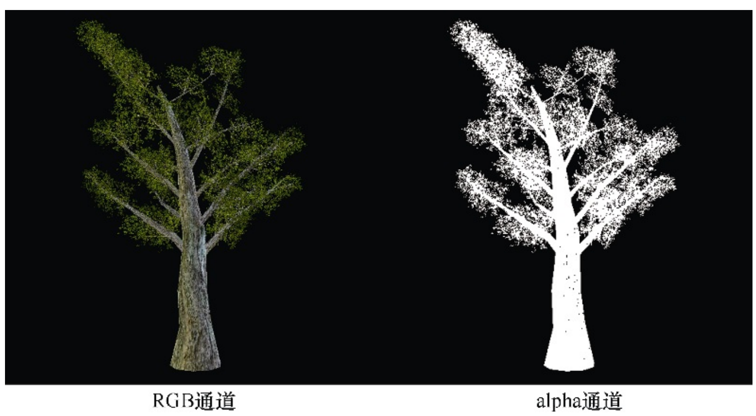


图12.2　一颗树木公告牌纹理及其alpha通道

假设y轴指向正上方，且平面xz表示地面，则树木公告牌通常被置于xz平面内并与y轴对齐而面向摄像机。图12.3以鸟瞰视角展示了几个公告牌的局部坐标系——可以看出，所有的公告牌都正在面对着摄像机搔首弄姿地“抢镜”。

在世界空间中，若给定一公告牌的中心位置（也就是四边形的中心

点）为$\boldsymbol{C}=(C_x,C_y,C_z)$，而摄像机的位置为$\boldsymbol{E}=(E_x,E_y,E_z)$，那么，这些信息就足以表示出该公告牌局部坐标系与世界空间的相对关系：

$$\boldsymbol{w}=\frac{(E_x-C_x,0,E_z-C_z)}{\|E_x-C_x,0,E_z-C_z\|}$$
$$\boldsymbol{v}=(0,\,1,\,0)$$
$$\boldsymbol{u}\;=\;\boldsymbol{v}\times\boldsymbol{w}$$

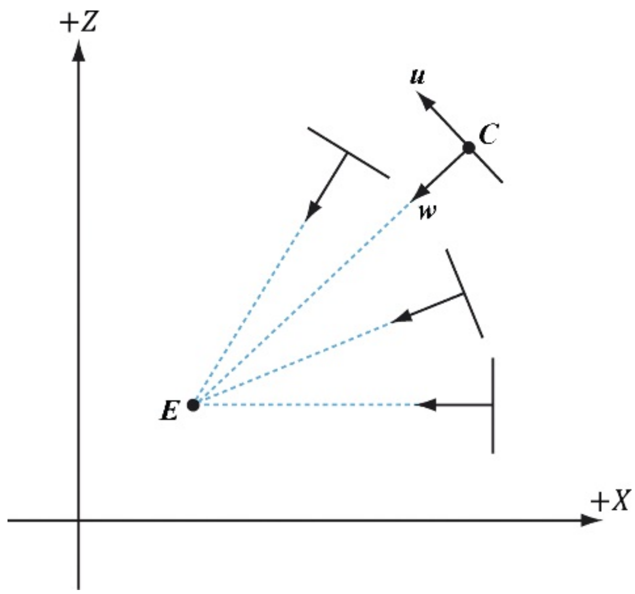


图12.3　面对摄像机的众公告牌

给出公告牌局部坐标系与世界空间的相对关系以及公告牌的在世界空间中的大小（world size），我们就能以下列代码来获得公告牌四边形的4个顶点坐标（见图12.4）：

```
v[0] = float4(gin[0].CenterW + halfWidth*right - halfHeight*up, 1.0f);
v[1] = float4(gin[0].CenterW + halfWidth*right + halfHeight*up, 1.0f);
v[2] = float4(gin[0].CenterW - halfWidth*right - halfHeight*up, 1.0f);
v[3] = float4(gin[0].CenterW - halfWidth*right + halfHeight*up, 1.0f);
```

注意，由于公告牌彼此之间的局部坐标系并不一致，所以每一个公告牌的四边形都要分别计算。

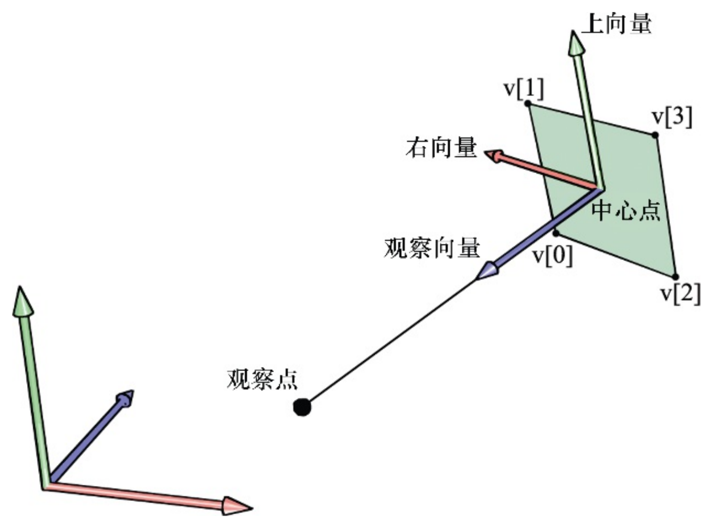


图12.4　根据公告牌的局部坐标系及其位于世界空间中的大小来计算公告牌四边形的顶点

对于眼下这个演示程序而言，我们将构造一系列距陆地表面有着特定距离的点图元（将PSO中的 PrimitiveTopologyType成员指定为D3D12_PRIMITIVE_TOPOLOGY_TYPE_POINT，并把ID3D12GraphicsCommandList::IASetPrimitiveTopology函数的参数指定为D3D_PRIMITIVE_TOPOLOGY_POINTLIST）。这些点表示的就是我们希望绘制的公告牌的中心点。在几何着色器中，我们既要把这些点扩展为公告牌的四边形，又要计算公告牌的世界矩阵。图12.5所示的就是该演示程序的效果。

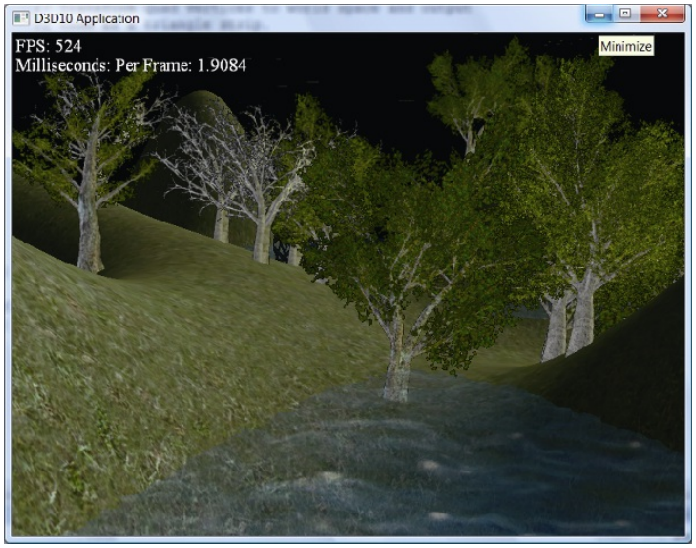


图12.5　树木公告牌例程的效果图

从图12.5中也能看出，这个示例源于第10章中的“Blend Demo”演示程序。

注意

一种通过CPU来实现公告牌技术的常用方法是，在动态顶点缓冲区（即上传堆）中来控制每个公告牌的4个顶点。每当摄像机移动时，这些顶点都会经CPU重新计算而随之更新，再通过`memcpy`函数将其复制到GPU端的缓冲区，以此令公告牌保持面对摄像机。若使用此方法，就一定要将每个公告牌的4个顶点都提交到IA（Input Assembler，输入装配器）阶段，并且还需更新动态顶点缓冲区，这会产生一定的开销。若采用几何着色器来实现公告牌技术，我们就能从容运用静态的顶点缓冲区，这是因为单凭几何着色器即可扩展公告牌的四边形，并令公告牌面

对摄像机。另外，几何着色器方案中公告牌所占用的内存空间也极小，因为创建每个公告牌只需向IA阶段提交一个顶点即可。

12.2.2　顶点结构体

我们用以下顶点结构体来描述公告牌：

```
struct TreeSpriteVertex
{
  XMFLOAT3 Pos;
  XMFLOAT2 Size;
};

mTreeSpriteInputLayout =
{
  { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
  { "SIZE", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
    D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
};
```

顶点结构体中所存储的点即表示在世界空间内公告牌的中心位置。除此之外，它还包含一个存有公告牌宽度/高度的数据成员，以表示其大小（已按世界空间的单位进行缩放）。因此，几何着色器便能知晓扩展后的公告牌会有多大（见图12.6）。通过调整每个顶点结构体中表示大小的成员，我们就可以方便地构造出不同尺寸的公告牌。

除了纹理数组（12.3节）以外，“Tree Billboards”演示程序中的其他C++代码部分都已有了具体的Direct3D编码实现（即创建顶点缓冲区、效果文件[5]与调用绘制方法等功能）。这样一来，我们只需将注意力转

移到TreeSprite.hlsl文件上即可。

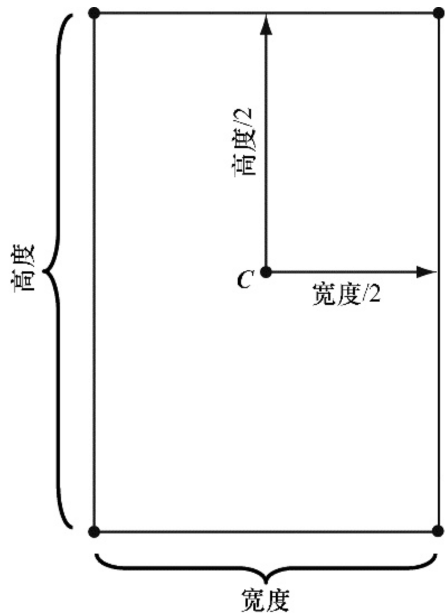


图12.6　将一个点扩展为一个四边形

12.2.3　HLSL文件

由于这是本书第一次向读者展示几何着色器示例，所以我们将在此列出完整的HLSL代码，使读者便于了解几何着色器、顶点着色器和像素着色器这三者的搭配使用方法。该效果文件也引入了一些之前未曾讨论过新对象（`SV_PrimitiveID`与`Texture2DArray`），在后续小节里会依次对此进行介绍。眼下，我们主要关注的是几何着色器程序`GS`。正如12.2.1节中所述，该着色器的功能是把一个点扩展为一个四边形，并使之与y轴对齐以正对摄像机。

```
//***************************************************************************
****
// TreeSprite.hlsl的作者为Frank Luna (C) 2015 版权所有
```

```
//*****************************************************************************
****

// 光源数量的默认值
#ifndef NUM_DIR_LIGHTS
  #define NUM_DIR_LIGHTS 3
#endif

#ifndef NUM_POINT_LIGHTS
  #define NUM_POINT_LIGHTS 0
#endif

#ifndef NUM_SPOT_LIGHTS
  #define NUM_SPOT_LIGHTS 0
#endif

// 包含光照所需的结构体与函数
#include "LightingUtil.hlsl"

Texture2DArray gTreeMapArray : register(t0);

SamplerState gsamPointWrap    : register(s0);
SamplerState gsamPointClamp    : register(s1);
SamplerState gsamLinearWrap    : register(s2);
SamplerState gsamLinearClamp   : register(s3);
SamplerState gsamAnisotropicWrap : register(s4);
SamplerState gsamAnisotropicClamp : register(s5);

// 每一帧都在变化的常量数据
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorld;
  float4x4 gTexTransform;
};

// 绘制过程中所用的杂项常量数据
cbuffer cbPass : register(b1)
{
  float4x4 gView;
  float4x4 gInvView;
  float4x4 gProj;
  float4x4 gInvProj;
  float4x4 gViewProj;
  float4x4 gInvViewProj;
  float3 gEyePosW;
  float cbPerObjectPad1;
  float2 gRenderTargetSize;
  float2 gInvRenderTargetSize;
```

```
  float gNearZ;
  float gFarZ;
  float gTotalTime;
  float gDeltaTime;
  float4 gAmbientLight;

  float4 gFogColor;
  float gFogStart;
  float gFogRange;
  float2 cbPerObjectPad2;

  // 对于每个以MaxLights为光源数量最大值的对象来说，索引[0,NUM_DIR_LIGHTS)表示的是方向光
  // 源，索引[NUM_DIR_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHTS]表示的是点光源
  // 索引[NUM_DIR_LIGHTS+NUM_POINT_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHT+
  // NUM_SPOT_LIGHTS)表示的则为聚光灯光源
  Light gLights[MaxLights];
};

// 每种材质都各有区别的常量数据
cbuffer cbMaterial : register(b2)
{
  float4  gDiffuseAlbedo;
  float3  gFresnelR0;
  float   gRoughness;
  float4x4 gMatTransform;
};

struct VertexIn
{
  float3 PosW : POSITION;
  float2 SizeW : SIZE;
};

struct VertexOut
{
  float3 CenterW : POSITION;
  float2 SizeW  : SIZE;
};

struct GeoOut
{
  float4 PosH   : SV_POSITION;
  float3 PosW   : POSITION;
  float3 NormalW : NORMAL;
  float2 TexC   : TEXCOORD;
  uint  PrimID : SV_PrimitiveID;
```

```
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout;

  // 直接将数据传入几何着色器
  vout.CenterW = vin.PosW;
  vout.SizeW  = vin.SizeW;

  return vout;
}

// 由于我们要将每个点都扩展为一个四边形（即4个顶点），因此每次调用几何着色器最多
输出4个顶点
[maxvertexcount(4)]
void GS(point VertexOut gin[1],
    uint primID : SV_PrimitiveID,
    inout TriangleStream<GeoOut> triStream)
{
  //
  // 计算精灵[6]的局部坐标系与世界空间的相对关系，以使公告牌与y轴对齐且面向观察者
  //

  float3 up = float3(0.0f, 1.0f, 0.0f);
  float3 look = gEyePosW - gin[0].CenterW;
  look.y = 0.0f; // 与y轴对齐，以此使公告牌立于xz平面
  look = normalize(look);
  float3 right = cross(up, look);

  //
  // 计算世界空间中三角形带的顶点（即四边形）
  //
  float halfWidth = 0.5f*gin[0].SizeW.x;
  float halfHeight = 0.5f*gin[0].SizeW.y;

  float4 v[4];
  v[0] = float4(gin[0].CenterW + halfWidth*right - halfHeight*up, 1.0f);
  v[1] = float4(gin[0].CenterW + halfWidth*right + halfHeight*up, 1.0f);
  v[2] = float4(gin[0].CenterW - halfWidth*right - halfHeight*up, 1.0f);
  v[3] = float4(gin[0].CenterW - halfWidth*right + halfHeight*up, 1.0f);

  //
  // 将四边形的顶点变换到世界空间，并将它们以三角形带的形式输出
  //

  float2 texC[4] =
```

```
  {
    float2(0.0f, 1.0f),
    float2(0.0f, 0.0f),
    float2(1.0f, 1.0f),
    float2(1.0f, 0.0f)
  };

  GeoOut gout;
  [unroll]
  for(int i = 0; i < 4; ++i)
  {
    gout.PosH   = mul(v[i], gViewProj);
    gout.PosW   = v[i].xyz;
    gout.NormalW = look;
    gout.TexC   = texC[i];
    gout.PrimID  = primID;

    triStream.Append(gout);
  }
}

float4 PS(GeoOut pin) : SV_Target
{
  float3 uvw = float3(pin.TexC, pin.PrimID%3);
  float4 diffuseAlbedo = gTreeMapArray.Sample(
    gsamAnisotropicWrap, uvw) * gDiffuseAlbedo;

#ifdef ALPHA_TEST
  // 忽略纹理alpha值 < 0.1的像素。这个测试要尽早完成，以便提前退出着色器，使满足
此条件的像素
  // 跳过着色器中不必要的后续处理流程
  clip(diffuseAlbedo.a - 0.1f);
#endif

  // 对法线插值可能导致其非规范化，因此需再次对它进行规范化处理
  pin.NormalW = normalize(pin.NormalW);

  // 光线经表面上一点反射到观察点这一方向上的向量
  float3 toEyeW = gEyePosW - pin.PosW;
  float distToEye = length(toEyeW);
  toEyeW /= distToEye; // 规范化处理

  // 光照项
  float4 ambient = gAmbientLight*diffuseAlbedo;

  const float shininess = 1.0f - gRoughness;
  Material mat = { diffuseAlbedo, gFresnelR0, shininess };
```

```
  float3 shadowFactor = 1.0f;
  float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
    pin.NormalW, toEyeW, shadowFactor);

  float4 litColor = ambient + directLight;

#ifdef FOG
  float fogAmount = saturate((distToEye - gFogStart) / gFogRange);
  litColor = lerp(litColor, gFogColor, fogAmount);
#endif

  // 从漫射反照率中获取alpha值的常用手段
  litColor.a = diffuseAlbedo.a;

  return litColor;
}
```

12.2.4 SV_PrimitiveID语义

在以上示例中，几何着色器内含有一个使用SV_PrimitiveID语义的特殊无符号整数参数。

```
[maxvertexcount(4)]
void GS(point VertexOut gin[1],
    uint primID:SV_PrimitiveID,
    inout TriangleStream<GeoOut> triStream)
```

若指定了该语义，则输入装配器阶段会自动为每个图元生成图元ID。在绘制n个图元的调用执行过程中，第一个图元被标记为0，第二个图元被标识为1，并以此类推，直到此绘制调用过程中最后一个图元被标记为$n-1$为止。对于单次绘制调用来说，其中的图元ID都是唯一的。在公告牌示例中，几何着色器不会用到此ID（尽管它可以使用），它仅将图元ID写入到输出的顶点之中，以此将它们传递到像素着色器阶段。而像素着色器则把图元ID用作纹理数组的索引，这是下一小节将讲述的

内容。

注意 Note

如果代码中不存在几何着色器，我们便可以将图元ID参数加入像素着色器的参数列表：

```
float4 PS(VertexOut pin, uint primID : SV_PrimitiveID) : SV_Target
{
  // 像素着色器的具体实现
}
```

但是，若代码内具有几何着色器，则图元ID必首先存在于其签名的参数之中[7]。继而几何着色器就能使用图元ID或将其传至像素着色器阶段（抑或同时执行这两种操作）。

注意 Note

输入装配器也能够生成顶点ID。为了实现这一点，我们需要向顶点着色器签名额外添加一个由语义`SV_VertexID`修饰的`uint`类型的参数：

```
VertexOut VS(VertexIn vin, uint vertID : SV_VertexID)
{
  // 顶点着色器的主体
```

```
}
```

此时，对于绘制方法`DrawInstanced`来说，其调用过程中的顶点ID将被标记为$0, 1..., n-1$，这里的n即为本次绘制调用中的顶点数量。而对于`DrawIndexedInstanced`绘制方法而言，其顶点ID则对应于顶点的索引值。

12.3 纹理数组

12.3.1 概述

顾名思义，纹理数组即为存放纹理的数组。就像所有的资源（纹理与缓冲区）一样，在C++代码中，纹理数组也由ID3D12Resource接口来表示。当创建ID3D12Resource对象时，可以通过设置DepthOrArraySize属性来指定纹理数组所存储的元素个数（对于3D纹理来说，此项设定的则为深度值）。我们在d3dApp.cpp文件中创建深度/模板纹理时，总是将该值设为1。如果查看Common/DDSTextureLoader.cpp中的CreateD3DResources12函数，便会明白这些代码究竟是如何来创建纹理数组与体纹理（volume texture）的。在HLSL文件中，纹理数组是通过Texture2DArray类型来表示的：

```
Texture2DArray gTreeMapArray;
```

现在，我们必须搞清楚为什么要使用纹理数组，而不是像下面那样做：

```
Texture2D TexArray[4];
...
float4 PS(GeoOut pin) : SV_Target
{
float4 c = TexArray[pin.PrimID%4].Sample(samLinear, pin.Tex);
```

在着色器模型5.1（Direct3D 12所对应的新模型版本）中确实可以这样写，但这在之前的Direct3D版本中却是行不通的。再者，以这种方

式来索引纹理可能会依具体硬件而产生少量开销，所以本章我们将使用纹理数组。

12.3.2　对纹理数组进行采样

在公告牌示例中，我们用以下代码对纹理数组进行采样[8]：

```
float3 uvw = float3(pin.TexC, pin.PrimID%4);
float4 diffuseAlbedo = gTreeMapArray.Sample(
  gsamAnisotropicWrap, uvw) * gDiffuseAlbedo;
```

使用纹理数组共需要3个坐标值：前两个坐标就是普通的2D纹理坐标，第三个坐标则是纹理数组的索引。例如，0、1、2分别是数组中的第一个、第二个以及第三个纹理的索引，后面的索引则以此类推。

在公告牌示例中，我们采用的是一个具有4个纹理元素的纹理数组，且每个元素中都存有独特的树木纹理（见图12.7）。但是，由于每次绘制调用所画出的树木要多于4棵，所以图元ID势必也要大于3。因此，我们对图元ID进行模4运算（`pin.PrimID % 4`），将其值映射为0、1、2或3，继而使数组元素拥有合理的数组索引。



图12.7　树木公告牌所用的图片

纹理数组的优点之一是可以在一次绘制调用过程中，画出一系列具有不同纹理的图元。一般来说，我们必须用不同的纹理来对每一个有着独立渲染项的网格进行处理：

```
SetTextureA();
DrawPrimitivesWithTextureA();

SetTextureB();
DrawPrimitivesWithTextureB();

...

SetTextureZ();
DrawPrimitivesWithTextureZ();
```

每当设置纹理或绘制调用的时候都会相应地产生一些开销。但使用了纹理数组，我们就能将设置纹理与绘制调用的过程都减少到一次：

```
SetTextureArray();
DrawPrimitivesWithTextureArray();
```

12.3.3　加载纹理数组

位于Common/DDSTextureLoader.h/.cpp文件中的代码，可以加载存有纹理数组的DDS文件。因此，重点就落在了创建含有纹理数组的DDS文件上。为此，我们需要使用微软公司所提供的texassemble工具。通过下列语法，此工具可以将t0.dds、t1.dds、t2.dds与t3.dds这4个图像合并成一个名为treeArray.dds的纹理数组：

```
texassemble -array -o treeArray.dds t0.dds t1.dds t2.dds t3.dds
```

注意，在用texassemble程序构建纹理数组时，输入的多个图像只能各有一种mipmap层级。经texassemble创建的纹理数组，可以再根据需求通过texconv工具生成多种mipmap层级，并改变纹理的格式：

```
texconv -m 10 -f BC3_UNORM treeArray.dds
```

12.3.4　纹理子资源

前面刚刚讨论过纹理数组，下面来谈一谈它的子资源（subresource）。图12.8展示了一个拥有若干纹理的纹理数组，其中的纹理还各有自己的mipmap链。Direct3D API用术语**数组切片**（array slice）来表示纹理数组中的某个纹理及其mipmap链，又用术语**mip切片**（mip slice）来表示纹理数组中特定层级的所有mipmap。子资源则是指纹理数组中某纹理的单个mipmap层级。

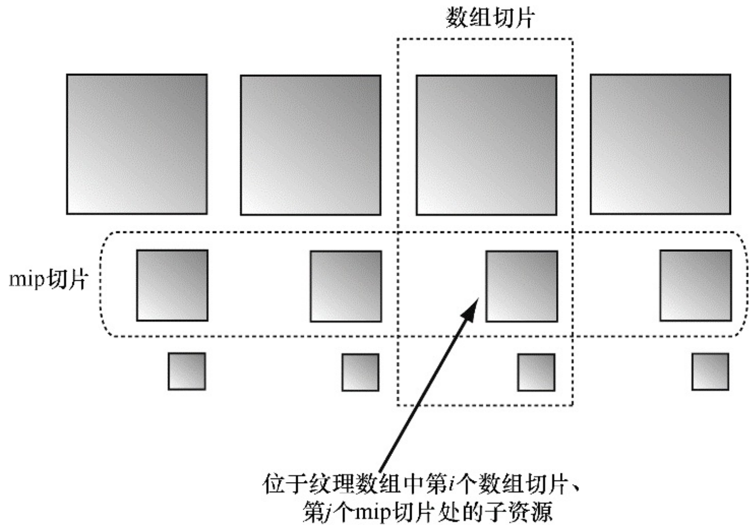


图12.8　有着4个纹理的纹理数组。每个纹理都有3种mipmap层级

这就是说，若给出纹理数组的索引以及mipmap层级，我们就能访问纹理数组中的相应子资源。值得注意的是，子资源也是由线性索引来标记的，而Direct3D所用的线性索引规则如图12.9所示。

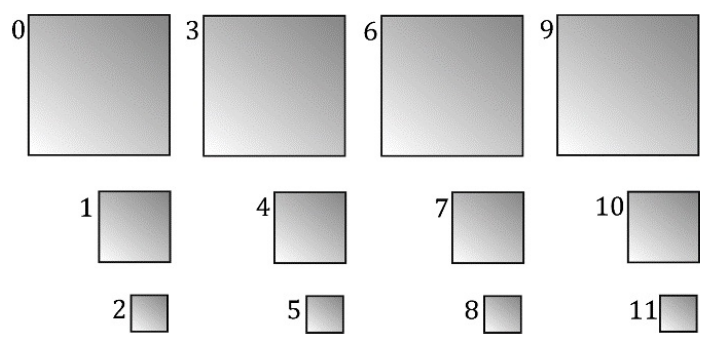


图12.9　在纹理数组中由线性索引来标记的子资源

下面的实用工具函数可根据给出的mip切片索引、数组切片索引、平面切片[9]索引、mipmap层级以及纹理数组的大小，直接计算出子资源的线性索引：

```
inline UINT D3D12CalcSubresource( UINT MipSlice, UINT ArraySlice,
  UINT PlaneSlice, UINT MipLevels, UINT ArraySize )
```

```
{
  return MipSlice + ArraySlice * MipLevels + PlaneSlice * MipLevels * Arra
ySize;
}
```

12.4 alpha-to-coverage技术

在运行“Tree Billboard”演示程序时，若以特定距离观察树木公告牌，便会发现其裁剪的边缘部分呈锯齿状。这个问题出在`clip`函数上，我们用它来遮罩不属于树木纹理的像素。此裁剪函数负责像素的生杀大权，也导致了树木边缘的过渡不够平滑。观察者与公告牌之间的距离也趁机推波助澜，因为两者距离过近会引起纹理放大（magnification）的发生，继而使块状失真更加明显。不仅如此，近距离还将导致低分辨率mipmap层级的启用。

解决此问题的方法之一是以透明混合取代alpha测试。通过线性纹理过滤，使边缘像素稍显模糊，从而促使由白（不透明的像素）至黑（被遮罩的像素）的过渡更为平滑。即透明混合将使公告牌图像边缘的不透明像素到被遮罩像素之间实现平滑的渐变。可是，运用透明混合需要将场景中的物体按从后至前的顺序进行排序与渲染。对为数不多的树木公告牌排序用不了多少开销，但若渲染的是一片树林或一片草原，则每帧中排序所付出的代价却是不容小觑的。不止于此，更糟糕的还要说是由后至前的排序过程，这将引发大量的重复绘制（overdraw，见第11章中的练习8）操作，会直接将我们的应用程序置于死地。

这时，不妨试一试MSAA（多重采样抗锯齿技术，multisampling antialiasing——参见4.1.7节），它可用于缓解多边形边沿的锯齿效果，从而使之相对平滑。该技术确实有效，但也带来了一些问题。MSAA会为每个像素逐一执行一次像素着色器，使像素着色器在像素的中心采

样，并基于可视性（visibility，每个子像素所执行的深度/模板测试结果）与覆盖情况（coverage，子像素的中心位于多边形的内部还是外部），将颜色信息共享给它的子像素（subpixel）。而更为关键的是，覆盖情况是在多边形层级（polygon level）上才确定下来的。因此，MSAA并不会检测alpha通道所定义的树木公告牌的裁剪边缘，而只是关注纹理所映射到四边形的边缘。有读者会问，那到底有没有办法通知Direct3D，使它在计算覆盖情况时考虑alpha通道这个因素呢？回答是肯定的，这便是被称之为“**alpha-to-coverage**”（有译作由透明至覆盖，等等）的技术。

在开启了MSAA与alpha-to-coverage后（令成员`D3D12_BLEND_DESC::AlphaToCoverageEnable = true`来实现），硬件就会检测像素着色器所返回的alpha值，并将其用于确定覆盖的情况[NVIDIA05]。例如，在使用4X MSAA时，若像素着色器返回的alpha值为0.5，则我们即可认为此像素里4个子像素中的2个位于多边形的范围之外，并据此来创建平滑的图像边缘。

一般来说，在以alpha遮罩的方式来裁剪树叶与围栏这类纹理时，建议总是使用alpha-to-coverage技术。当然，前提是需要开启MSAA。在演示程序中，可以通过`Set4xMsaaState`函数使示例框架创建出支持4X MSAA的后台缓冲区与深度缓冲区[10]。

12.5　小结

1．假设我们不使用曲面细分这个步骤，则几何着色器这个可选阶段将位于顶点着色器与像素着色器这两个阶段之间。几何着色器会针对输入装配器传来的每一个图元进行调用并执行处理。通过配置，它可以不输出图元，也能输出一个或一个以上的图元。而输出的图元类型，也可能与输入图元的类型并不相同。在输出的图元顶点离开几何着色器之前，应当将其变换到齐次裁剪空间。接下来，从几何着色器输出的图元，会进入渲染流水线的光栅化阶段。几何着色器应编写在效果文件中，并紧邻顶点着色器与像素着色器（处于二者之间）。

2．公告牌技术采用的是附有图像的四边形对象，可将它作为真实3D模型对象的一种替代品。对于远处的物体来说，公告牌足以以假乱真。若具有纹理的四边形能够满足用户的需求，则它的优点便是可以节省GPU渲染整个3D对象的处理时间。这项技术可用于渲染森林中的树木，离摄像机近的可用真正的3D几何体，而远处的树木用公告牌即可。为了使公告牌更加逼真，一定要令它总是面向摄像机。而且，在几何着色器中实现公告牌技术往往是最适合的。

3．我们可以把一种由语义`SV_PrimitiveID`修饰的特殊`uint`类型参数加入到几何着色器的参数列表之中，其示例代码如下：

```
[maxvertexcount(4)]
void GS(point VertexOut gin[1],
    uint primID : SV_PrimitiveID,
    inout TriangleStream<GeoOut> triStream);
```

若指定了此语义，它就会告知输入装配器环节自动为每个图元生成一个图元ID。当执行的绘制调用要画n个图元时，第一个图元被标记为0，第二个图元将标记为1，并以此类推，直至绘制调用中的最后一个图元被标记为$n-1$。如果用户没有使用几何着色器，则应当将图元ID参数添加至像素着色器的参数列表内。但是，若使用了几何着色器，那么一定要把图元ID参数设置在几何着色器的签名当中。接下来，几何着色器就会运用此图元ID，或将其传入像素着色器阶段（抑或同时执行这两种操作）。

4．输入装配器阶段可以生成顶点ID。为此，我们需要向顶点着色器签名添加额外的`uint`类型参数，并为它辅以`SV_VertexID`语义。对于`DrawInstanced`函数来说，此绘制调用中的顶点ID将依次被标记为：0, 1, …, $n-1$，其中n为此调用过程中的顶点个数。而针对`DrawIndexedInstanced`函数而言，其顶点ID则对应于顶点的索引值。

5．见名知意，纹理数组即存有纹理的数组。在C++代码中，纹理数组就像其他的资源一样（纹理与缓冲区）用`ID3D12Resource`接口来表示。在创建`ID3D12Resource`对象时，`DepthOrArraySize`就是用于指定纹理数组中元素个数的属性（对于3D纹理来说，它指定的是资源的深度值）。而在HLSL中，纹理数组由类型`Texture2DArray`来表示。使用纹理数组时，共需要三个纹理坐标。前两个坐标是普通的2D纹理坐标，第三个坐标则是纹理数组中的索引。例如，0、1、2分别为数组中前3个纹理的索引，后面的则依此类推。使用纹理数组的优点之一是：我们可以在单次绘制调用的过程中，以多种不同的纹理渲染出一系

列图元。而每个图元都有一个指向纹理数组的索引，它指定了图元所应用的纹理。

6．在确定子像素的覆盖情况时，alpha-to-coverage可指挥硬件检查像素着色器所返回的alpha数据。开启这项技术可以使树叶和围栏这样的alpha遮罩裁剪纹理具有平滑的边缘。Alpha-to-coverage的开启与否由PSO中的`D3D12_BLEND_DESC::AlphaToCoverageEnable`字段来决定。

12.6　练习

1．思考一下，如何用线条带在xz平面内绘制一个圆形，以及如何运用几何着色器将此线条带扩展为一个不带上下底的圆柱体。

2．正二十面体是一种粗略近似于球体的几何图形。通过对每个三角形进行细分（见图12.10），并在球面上投影新生成的顶点，我们就能获得更逼近于球体的几何体（由于所有单位向量的首部都在单位球面之上，因此，可以将顶点投影到单位球面上的操作简单地视为对位置向量进行规范化处理）。在这个练习当中，先要构建并渲染出一个正二十面体。再根据它与摄像机之间的距离d，用几何着色器对此正二十面体逐步进行细分。例如，如果$d < 15$，则对原始的正二十面体细分两次；若$15 \leqslant d \leqslant 30$，就对初始的正二十面体执行一次细分；当$d \geqslant 30$时，直接渲染初始的正二十面体即可。整体的思路就是：仅在物体距离摄像机较近的时候，才使用数量较多的多边形进行渲染，使其表现得更为细腻；若对象离观察位置较远，则使用较为粗糙的网格即可，我们也无需浪费GPU资源去处理本就看不清的物体细节。图12.10分别以线框模式与实体（光照）模式，接连展示了3种多面体的LOD（细节级别level-of-detail）。此习题中所用的曲面细分与正二十面体知识可回顾7.4.3节中的相关讨论。

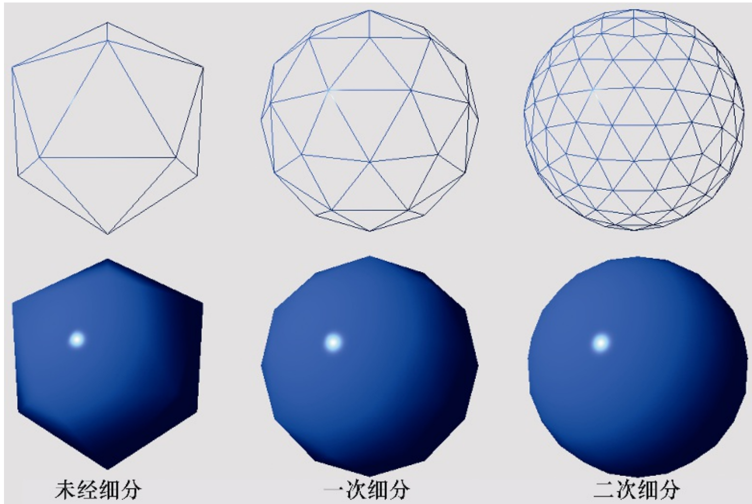


图12.10　通过将顶点投影到单位球面上的方法对正二十面体进行细分

3．通过令三角形随时间函数沿其平面法线的方向进行平移，我们就能模拟出简单的爆炸效果。这个模拟过程可在几何着色器中实现。针对每个输入到几何着色器的三角形来说，先以着色器计算其平面法线$\boldsymbol{n}$。待爆炸开始后，沿方向$\boldsymbol{n}$平移的三角形顶点$\boldsymbol{p}_0$、$\boldsymbol{p}_1$和$\boldsymbol{p}_2$在时刻t的位置分别为：

$$\boldsymbol{p}_i' = \boldsymbol{p}_i + t\boldsymbol{n} \quad 其中 \quad i = 0, 1, 2$$

平面法线$\boldsymbol{n}$不一定是单位长度，可对其进行缩放以控制爆炸的速度。我们甚至可以根据图元ID来调节平面法线的长度，以使每个图元都能按不同的弹射速度四处飞溅。用一个正二十面体（不需要细分）作为实验的网格来实现爆炸的效果。

4．令网格的顶点法线可视化通常会更便于调试。编写一个效果文件，将网格的顶点法线都渲染为短线段。为了做到这一点，我们需要实现一个以网格的点图元作为输入数据的几何着色器（即以图元拓

扑D3D_PRIMITIVE_TOPOLOGY_POINTLIST来表示网格中的诸顶点），从而将每个顶点传至几何着色器阶段。这时，该几何着色器便能够将每个点扩展为长度为L的线段。若顶点的位置为$\boldsymbol{p}$且法线为$\boldsymbol{n}$，则其对应的2个线段端点就能分别表示为$\boldsymbol{p}$与$\boldsymbol{p}+L\boldsymbol{n}$。待上述内容能够顺利实现之后，先照常绘制一遍网格，后以法向量可视化技术再渲染一遍场景，以此令法线绘制在场景的上面，便于观察。对了，这个练习要以“Blend Demo”（混合）演示程序作为测试场景。

5．本练习与上一个练习比较相似：编写一个将网格的平面法线渲染为短线段的效果文件。对此效果文件而言，几何着色器应以三角形作为输入数据，并计算其平面法线及输出短线段。

6．本练习将展示：对于DrawInstanced方法而言，其绘制调用过程中的顶点ID会以$0,1,...,n-1$来加以标记，这里的n为绘制调用过程中所用的顶点个数；而针对DrawIndexedInstanced方法调用来说，顶点ID则对应于其顶点的索引值。

按下列方式来修改“Tree Billboards”演示程序。首先，将顶点着色器的代码改写为：

```
VertexOut VS(VertexIn vin, uint vertID : SV_VertexID)
{
  VertexOut vout;

  // 直接将数据传递给几何着色器
  vout.CenterW = vin.PosW;
  vout.SizeW  = float2(2+vertID, 2+vertID);

  return vout;
}
```

换句话说，这段代码将基于树木公告牌的中心，按顶点ID的数值来对公告牌进行缩放。直接运行程序，在绘制完16个公告牌时，它们的大小应当在范围为2～17。现在再来修改公告牌的绘制方法，之前的单次绘制调用一次性可渲染16个点，而本次则以4次DrawInstanced调用来加以取代，就像这样：

```
  if(ritems[i]->Geo->Name=="treeSpritesGeo"){
    cmdList->DrawInstanced(4,1,0,0);
    cmdList->DrawInstanced(4,1,4,0);
    cmdList->DrawInstanced(4,1,8,0);
    cmdList->DrawInstanced(4,1,12,0);
  }else{
    cmdList->DrawIndexedInstanced(ri->IndexCount,1,ri->StartIndexLocation
,ri->
          BaseVertexLocation.0);
```

再次运行程序……这一次，公告牌的大小会处于范围为2～5。这是因为每次绘制调用会画出4个顶点，因此每次调用过程中的顶点ID范围为0～3。最后再尝试借助一个索引缓冲区以及4次DrawIndexedInstanced调用绘制树木公告牌。运行程序之后，便会发现公告牌的大小又重新回到范围2～17。这是因为在使用DrawIndexedInstanced方法时，顶点ID将对应于顶点的索引值。

7．通过下列方式来修改“Tree Billboards”演示程序。首先，从像素着色器中去掉“模4运算”：

```
  float3 uvw = float3(pin.TexC, pin.PrimID);
```

现在再来运行程序。由于我们共绘制了16个图元，图元ID的范围为0～15，因而这些ID会超出像素数组的边界。然而，这实际上并不会导

致错误的发生，因为越界的索引值会被钳制为最大的有效索引（这里即为3）。再按如下方法以4次DrawInstanced绘制调用取代之前的一次性绘制16个点：

```
   if(ritems[i]->Geo->Name=="treeSpritesGeo"){
     cmdList->Drawlnstanced(4,1,0,0);
     cmdList->Drawlnstanced(4,1,4,0);
     cmdList->Drawlnstanced(4,1,8,0);
     cmdList->Drawlnstanced(4,1,12,0);
   }else{
     cmdList->Drawlndexedlnstanced(ri->lndexCount,1,ri->Startlndexlocation
,ri->
           BaseVertexLocation.0);
```

再次运行程序。因为每次DrawInstanced调用共绘制4个图元，所以每次绘制调用过程中的图元ID范围为0～3，也就不用再进行钳位操作了。这样一来，即可将图元ID用作索引，还不会发生越界的情况。这个示例充分展示了在每次绘制调用时，图元ID的“计数”都会被重置为0的现象。

---

[1]　文献中给出的例子是，如果顶点结构体中定义了“float3 pos: POSITION;”与“float2 tex: TEXCOORD;”这两项成员，意即每个顶点元素中含有5个标量。假设此时将maxvertexcount设置为4，则几何着色器每次输出20个标量，以峰值性能执行。此文献公布于2008年，这段描述针对的GPU型号为GeForce 8800 GTX。

[2]　RestartStrip函数表示结束当前三角形带的绘制，下面绘制另一个三角形带。每3个顶点调用一次RestartStrip表示每3个顶点组成一个三角形带，也就是一个三角形列表。

[3]　例如，如果在处理三角形带中的最后一个三角形时，哎呀，少了个顶点，那么这最后一个三角形就默认失踪了……

[4]　玩过《暴力摩托》的读者一定会明白作者在说什么……

[5]　原文为effects，即效果。DirectX 12之前的版本提供了一种已开源的“效果框架”（effects framework），使用户可以在运行时更便捷地管理渲染流水线的状态、HLSL着色器以及运行时变量等。但在DirectX 12中，这套框架已不复存在。本书中还存有此用语若干处，现都译为效果文件，其意与HLSL文件等价。

[6]　精灵，sprite。通常来讲，是一种不经渲染流水线而直接绘制到渲染目标的2D位图。公告牌实为应用于3D环境中的精灵，是具有`alpha`通道且面向摄像机的图像。DirectX 8至DirectX 10专门提供了一组绘制精灵的相关函数，但从DirectX 11开始，这些函数便被取消了。如今，实现精灵的方法可参见《Multiple Ways to Render Point Sprites in DX11》与《Sprites and textures》。

[7]　简言之，若存在几何着色器，则几何着色器中首获图元ID；若不存在几何着色器，则像素着色器首获图元ID。此二者谁离输入装配器阶段最近并被开启，谁先获得图元ID。

[8]　本书代码中实际写作“`pin.PrimID%3`”，如果dds文件中有4种树木图像的话，应当为“`pin.PrimID%4`”。

[9]　在Direct3D 12中，新引入了术语平面切片，即plane slice。它使用

户能将某些YUV平面格式（planar format）的分量以索引方式单独提取出来，以供使用。若程序不涉及此功能，可将参数`PlaneSlice`设置为0。

[10]　“Tree Billboard”演示程序默认为不开启MSAA，我们可以通过F2键进行切换。满心欢喜地按下F2键后……崩溃……（莫非我用的是假的DirectX 12？！）经查阅，最终在DirectXTK12的wiki文档Simple rendering中发现，Direct3D 12并不支持创建MSAA交换链（这与DirectX 12采用新式“flip”类型的交换链有关，MSAA仅支持旧式的“bit-blt”类型交换链，但在作者用的预览版里很可能还在支持直接创建MSAA交换链。关于新旧式交换链类型的讨论文章有很多，如《For best performance, use DXGI flip model》等）。解决的办法是用户要自行创建MSAA渲染目标。如果读者在运行时也发生同样的错误，可以尝试从这个方向着手。其实《Simple rendering》一文中已给出了MSAA的正确打开方式。现在，微软已对此专门提供了演示程序，可参见SimpleMSAA_UWP12例程。全书对UWP方面的编程只字未提，读者正好可借此机会做些了解）。

第13章　计算着色器

当今的GPU已经针对单址或连续地址的大量内存处理（亦称为流式操作，streaming operation）进行了优化，这与CPU面向内存随机访问的设计理念则刚好背道而驰[Boyd10]。再者，考虑到要对顶点与像素分别进行单独的处理，因此GPU现已经采用了大规模并行处理架构。例如，NVIDIA公司开发的“Fermi”架构最多可支持16个流式多处理器（streaming multiprocessor，SM），而每个流式处理器又均含有32个CUDA核心，也就是共512个CUDA核心[NVIDIA09]。

显然，图形的绘制优势完全得益于GPU架构，因为这架构就是专为绘图而精心设计的。但是，一些非图形应用程序同样可以从GPU并行架构所提供的强大计算能力中受益。我们将GPU用于非图形应用程序的情况称为通用**GPU**程序设计（通用GPU编程。General Purpose GPU programming，GPGPU programming）。当然，并不是所有的算法都适合由GPU来执行，只有数据并行算法（data-parallel algorithm）才能发挥出GPU并行架构的优势。也就是说，仅当拥有大量待执行相同操作的数据时，才最适宜采用并行处理。像素着色这种图像处理工作就是一种极好的示例，因为每个被绘制的像素片段都要经过像素着色器的统一处理。又如，查看前一章中模拟波浪的代码便会发现：在更新步骤中，我们需要针对每一个栅格元素都进行一遍相同的运算。因此，以GPU来执行这些计算工作也是不错的选择，这样一来，每个栅格元素都可以由

GPU并行地更新。粒子系统则是另一个实例，我们可简化粒子之间的关系模型，使它们彼此毫无关联，不会相互影响，以此使每个粒子的物理特征都可以分别独立地计算出来。

对于GPGPU编程而言，用户通常需要将计算结果返回CPU供其访问。这就需将数据由显存复制到系统内存，虽说这个过程的速度较慢（见图13.1），但是与GPU在运算时所缩短的时间相比却是微不足道的。针对图形处理任务来说，我们一般将运算结果作为渲染流水线的输入，所以无须再由GPU向CPU传输数据。例如，我们可以用计算着色器（compute shader）对纹理进行模糊处理（blur），再将着色器资源视图（shader resource view）与模糊处理后的纹理相绑定，以作为着色器的输入。

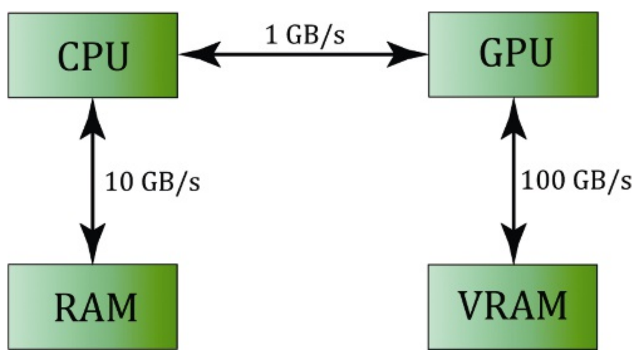


图13.1　根据[Boyd10]文献重新绘制的示意图。图中所示的是CPU与RAM（系统内存）、CPU与GPU以及GPU与VRAM（显存）之间的存储器带宽速度。其中所列的数字仅用于说明不同器件之间的传输带宽在数量级上的差别。显然，CPU与GPU之间的数据传输速度为整个系统的瓶颈

计算着色器虽然是一种可编程的着色器，但Direct3D并没有将它直接归为渲染流水线中的一部分。虽然如此，但位于流水线之外的计算着色器却可以读写GPU资源（见图13.2）。从本质上来说，计算着色器能

够使我们访问GPU来实现数据并行算法，而不必渲染出任何图形。正如前文所说，这一点即为GPGPU编程中极为实用的功能。另外，计算着色器还能实现许多图形特效——因此对于图形程序员来说，它也是极具使用价值的。前面提到，由于计算着色器是Direct3D的组成部分，也可以读写Direct3D资源，由此我们就可以将其输出的数据直接绑定到渲染流水线上。

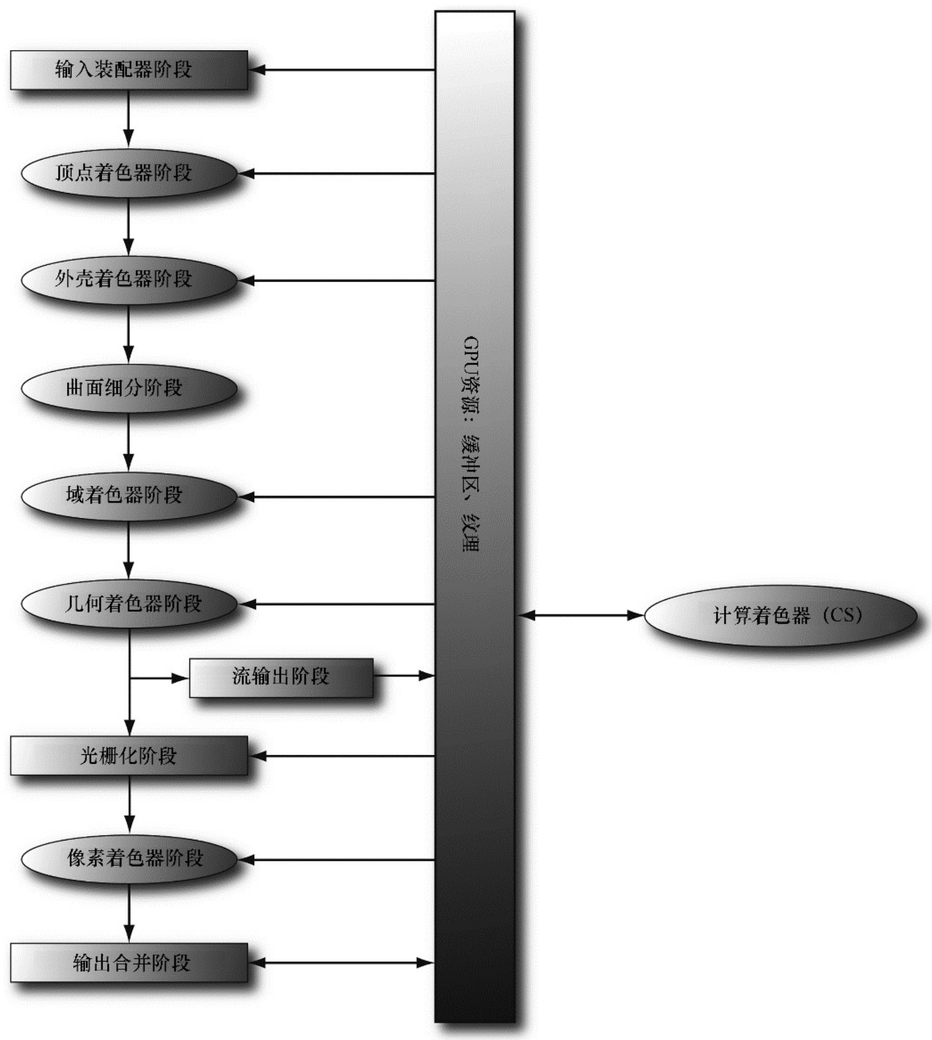


图13.2　计算着色器并非渲染流水线的组成部分，但是却可以读写GPU资源。而且计算着色器也可以参与图形的渲染或单独用于GPGPU编程

学习目标：

1．学习如何编写计算着色器。

2．对硬件处理线程组以及其中线程的方式等高级知识有一定的认识。

3．探究哪些Direct3D资源能被设置为计算着色器的输入与输出。

4．了解各种线程ID及其用法。

5．学习共享内存的相关知识，并知晓它为何可以用于优化性能。

6．探索怎样才能获取更多关于GPGPU编程的细节信息。

13.1　线程与线程组

在GPU编程的过程中，根据程序具体的执行需求，可将线程划分为由线程组（thread group）构成的网格（grid）。一个线程组运行于一个多处理器之上。因此，对于拥有16个多处理器的GPU来说，我们至少应将任务分解为16个线程组，以此令每个多处理器都充分地运转起来。但是，要获得更佳的性能，我们还应当令每个多处理器至少拥有两个线程组，使它能够切换到不同的线程组进行处理，以连续不停地工作[Fung10]（线程组在运行的过程中可能会发生停顿，例如，着色器在继续执行下一个指令之前会等待纹理的处理结果，此时即可切换至另一个线程组）。

每个线程组中都有一块共享内存[1]，供组内的线程访问。但是，线程并不能访问其他组中的共享内存。同理，同组内的线程间能够进行同步操作，不同组的线程间却不能实现这一点。事实上，我们也无法控制不同线程组间的处理顺序，因为这些线程组可能正运行在不同的多处理器上。

一个线程组中含有n个线程。硬件实际上会将这些线程分为多个**warp**（每个warp中有32个线程），而且多处理器会以SIMD32的方式（即32个线程同时执行相同的指令序列）来处理warp。每个CUDA核心都可处理一个线程，前面也提到了，“Fermi”架构中的每个多处理器都具有32个CUDA核心（因此，CUDA核心就像一条专设的SIMD“计算通道”（lane））。在Direct3D中，我们能够以非32的倍数值来指定线程组

的大小。但是出于性能的原因，我们应当总是将线程组的大小设置为warp尺寸的整数倍[Fung10]。

对于各种型号的图形硬件来说，线程数为256的线程组是一种普遍适于工作的初始设置。我们可以以此值为基础，再根据具体需求尝试将其调整为其他大小。值得注意的是，修改每个线程组中的线程数量也会对线程组的分派（dispatch，调度）次数产生影响[2]。

注意 Note

NVIDIA公司生产的图形硬件所用的warp单位共有32个线程。而ATI公司[3]采用的“wavefront”单位则具有64个线程，且建议为其分配的线程组大小应总为wavefront尺寸的整数倍[Bilodeau10]。另外，值得一提的是，不管是warp还是wavefront，它们的大小在未来几代中都有可能发生改变。

在Direct3D中可通过调用下列方法来启动线程组：

```
void ID3D12GraphicsCommandList::Dispatch(
 UINT ThreadGroupCountX,
 UINT ThreadGroupCountY,
 UINT ThreadGroupCountZ);
```

此方法可开启一个由线程组构成的3D网格，但是我们在本书中仅关注线程组2D网格。下面的调用示例会分派一个在x方向上为3、y方向

上为2，即总数为$3 \times 2 = 6$个线程组的网格（见图13.3）。

```
cmdList->Dispatch(3, 2, 1);
```

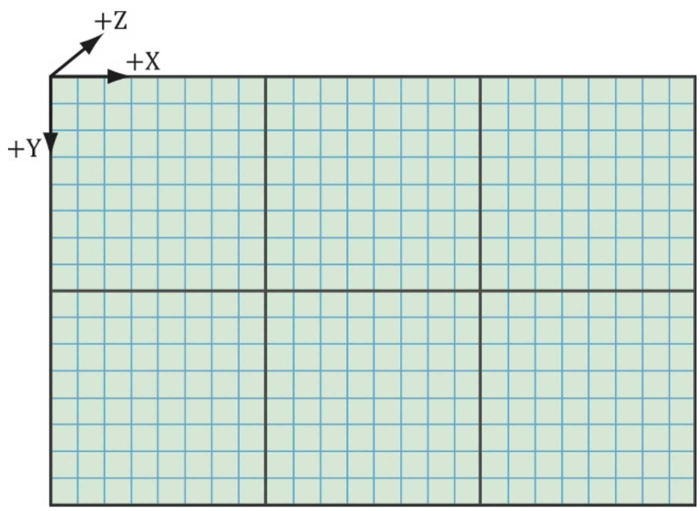


图13.3　分派一个规模为3×2的线程组。此例假设每个线程组都有8×8条线程

13.2　一个简单的计算着色器

以下是将两个纹理进行简单累加的计算着色器示例，假设所有的纹理都具有相同的大小。虽然该着色器有点索然无味，却五脏俱全，能详细地展示出计算着色器的基本套路语法。

```
cbuffer cbSettings
{
  // 计算着色器能访问的常量缓冲区数据
};

// 数据源及着色器的输出
Texture2D gInputA;
Texture2D gInputB;
RWTexture2D<float4> gOutput;

// 线程组中的线程数。组中的线程可以被设置为1D、2D或3D的网格布局
[numthreads(16, 16, 1)]
void CS(int3 dispatchThreadID : SV_DispatchThreadID) // 线程ID
{
  // 对两种源像素中横纵坐标分别为x、y处的纹素进行求和，并将结果保存到相应的gOutput 纹素中
  gOutput[dispatchThreadID.xy] =
    gInputA[dispatchThreadID.xy] +
    gInputB[dispatchThreadID.xy];
}
```

可见，一个计算着色器由下列要素构成：

1．通过常量缓冲区访问的全局变量。

2．输入与输出资源（此内容将在13.3节中讨论）。

3．[numthreads(X, Y, Z)]属性，指定3D线程网格中的线程数量。

4. 每个线程都要执行的着色器指令。

5. 线程ID系统值参数（在13.4节中讨论）。

不难看出，我们能够根据需求定义出不同的线程组布局。例如，可以定义一个具有X个线程的单行线程组`[numthreads(X, 1, 1)]`或内含Y个线程的单列线程组`[numthreads(1, Y, 1)]`。抑或通过将维度z设为1来定义规模为$X \times Y$的2D线程组，形如`[numthreads(X, Y, 1)]`。我们应结合所遇到的具体问题来选择适当的线程组布局。如同前一节中提到的那样：针对NVIDIA品牌的显卡来说，线程组中的总线程数应为warp大小（32）的整数倍，而ATI公司生产的显卡应为wavefront尺寸（64）的整数倍。又因wavefront大小的倍数（$64 \times n$）必为warp尺寸的倍数（$32 \times m$），因此，以前者的线程数为基础进行设置对两种显卡都适用。

计算流水线状态对象

为了开启计算着色器，我们还需使用其特定的“计算流水线状态描述”。此描述中的字段远少于`D3D12_GRAPHICS_PIPELINE_STATE_DESC`结构体。这是因为计算着色器位列图形流水线之外，因此所有的图形流水线状态都不适用于计算着色器，也就无须以此对它进行设置。下面给出一个创建计算流水线状态对象的示例[4]：

```
D3D12_COMPUTE_PIPELINE_STATE_DESC wavesUpdatePSO = {};
wavesUpdatePSO.pRootSignature = mWavesRootSignature.Get();
wavesUpdatePSO.CS =
{
```

```
  reinterpret_cast<BYTE*>(mShaders["wavesUpdateCS"]->GetBufferPointer()),
  mShaders["wavesUpdateCS"]->GetBufferSize()
};
wavesUpdatePSO.Flags = D3D12_PIPELINE_STATE_FLAG_NONE;
ThrowIfFailed(md3dDevice->CreateComputePipelineState(
  &wavesUpdatePSO, IID_PPV_ARGS(&mPSOs["wavesUpdate"])));
```

根签名定义了什么参数才是着色器所期望的输入（CBV、SRV等）。而`cs`（即compute shader的缩写）字段就是所指定的计算着色器。下列代码展示了一个将着色器编译为字节码的示例：

```
mShaders["wavesUpdateCS"] = d3dUtil::CompileShader(
  L"Shaders\\WaveSim.hlsl", nullptr, "UpdateWavesCS", "cs_5_0");
```

13.3　数据的输入与输出资源

能与计算着色器绑定的资源类型有缓冲区与纹理两种。我们已经用过诸如顶点、索引与常量这几类缓冲区，在第9章中也尝试过纹理资源。本节来讲述计算着色器所采用的输入、输出资源。

13.3.1　纹理输入

在前一节的计算着色器示例中，我们定义了两个输入纹理资源：

```
Texture2D gInputA;
Texture2D gInputB;
```

通过给输入纹理gInputA与gInputB分别创建SRV（着色器资源视图），再将它们作为参数传入根参数，我们就能令这两个纹理都绑定为着色器的输入资源。例如：

```
cmdList->SetComputeRootDescriptorTable(1, mSrvA);
cmdList->SetComputeRootDescriptorTable(2, mSrvB);
```

这个过程其实与着色器资源视图绑定到像素着色器的方法相同。但还需要注意的是，SRV都是只读资源。

13.3.2　纹理输出与无序访问视图

在前一节的计算着色器代码中，我们定义了一个输出资源：

```
RWTexture2D<float4> gOutput;
```

计算着色器处理输出资源的方式比较特殊，它们的类型还有一个特别的前缀“RW”，意为读与写。顾名思义，我们可以对计算着色器中的这类资源元素进行读写操作。相比之下，纹理gInputA与gInputB仅为只读属性。当然，别忘记用尖括号模板语法，如<float4>来指定输出资源的类型与维数。如果输出的是DXGI_FORMAT_R8G8_SINT类型的2D整型资源，则在HLSL文件里应当这样写：

```
RWTexture2D<int2> gOutput;
```

输出资源与输入资源的绑定方法是全然不同的。为了绑定在计算着色器中要执行写操作的资源，我们需要将其与称为无序访问视图（Unordered Access View，UAV）的新型视图关联在一起。在代码中，我们用描述符句柄来表示无序访问视图，且通过结构体D3D12_UNORDERED_ACCESS_VIEW_DESC来对它进行描述。创建这种视图的整个过程与着色器资源视图很相似。这里给出一个为纹理资源创建UAV的示例：

```
D3D12_RESOURCE_DESC texDesc;
ZeroMemory(&texDesc, sizeof(D3D12_RESOURCE_DESC));
texDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
texDesc.Alignment = 0;
texDesc.Width = mWidth;
texDesc.Height = mHeight;
texDesc.DepthOrArraySize = 1;
texDesc.MipLevels = 1;
texDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
texDesc.SampleDesc.Count = 1;
texDesc.SampleDesc.Quality = 0;
texDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
texDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS;
```

```
ThrowIfFailed(md3dDevice->CreateCommittedResource(
  &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
  D3D12_HEAP_FLAG_NONE,
  &texDesc,
  D3D12_RESOURCE_STATE_COMMON,
  nullptr,
  IID_PPV_ARGS(&mBlurMap0)));

D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING
;
srvDesc.Format = mFormat;
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Texture2D.MostDetailedMip = 0;
srvDesc.Texture2D.MipLevels = 1;

D3D12_UNORDERED_ACCESS_VIEW_DESC uavDesc = {};

uavDesc.Format = mFormat;
uavDesc.ViewDimension = D3D12_UAV_DIMENSION_TEXTURE2D;
uavDesc.Texture2D.MipSlice = 0;

md3dDevice->CreateShaderResourceView(mBlurMap0.Get(),
  &srvDesc, mBlur0CpuSrv);
md3dDevice->CreateUnorderedAccessView(mBlurMap0.Get(),
  nullptr, &uavDesc, mBlur0CpuUav);
```

从代码中可以看出，如果一个纹理需要与UAV相绑定，则此纹理必须用标志D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS来创建。在上面的示例中，我们将纹理分别绑定为一个UAV与一个SRV（但是两者却不能同时生效）。这是一种很常见的手段，因为我们通常会在计算着色器中对纹理执行某些操作（所以将纹理作为UAV绑定到计算着色器），而后还可能用此纹理对几何体进行贴图，因此需要再将它以SRV绑定到顶点着色器或像素着色器。

回顾一下，类型为D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV的描述符堆可以混合存放CBV（常量缓冲区视图）、SRV和UAV。因

此，我们就能将UAV描述符置于这种堆中。这些描述符一旦位于堆中，我们就能方便地将描述符句柄作为参数传至根参数，使资源绑定到流水线上，以供分派调用（dispatch call，调度调用）[5]使用。试考虑以下用于计算着色器的根签名：

```
void BlurApp::BuildPostProcessRootSignature()
{
  CD3DX12_DESCRIPTOR_RANGE srvTable;
  srvTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1, 0);

  CD3DX12_DESCRIPTOR_RANGE uavTable;
  uavTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_UAV, 1, 0);

  // 根参数可以是描述符表、根描述符或根常量
  CD3DX12_ROOT_PARAMETER slotRootParameter[3];

  // 提高程序性能的小窍门：按变更频率由高至低的顺序来填写根参数
  slotRootParameter[0].InitAsConstants(12, 0);
  slotRootParameter[1].InitAsDescriptorTable(1, &srvTable);
  slotRootParameter[2].InitAsDescriptorTable(1, &uavTable);

  // 根签名由一系列根参数构成
  CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(3, slotRootParameter,
    0, nullptr,
    D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

// 创建一个具有3个槽位的根签名，第一个指向常量缓冲区，第二个指向含有单个着色器资源视图的描述符
// 表，第三个指向含有单个无序访问视图的描述符表
  ComPtr<ID3DBlob> serializedRootSig = nullptr;
  ComPtr<ID3DBlob> errorBlob = nullptr;
  HRESULT hr = D3D12SerializeRootSignature(&rootSigDesc,
    D3D_ROOT_SIGNATURE_VERSION_1,
    serializedRootSig.GetAddressOf(), errorBlob.GetAddressOf());

  if(errorBlob != nullptr)
  {
    ::OutputDebugStringA((char*)errorBlob->GetBufferPointer());
  }
  ThrowIfFailed(hr);

  ThrowIfFailed(md3dDevice->CreateRootSignature(
    0,
```

```
    serializedRootSig->GetBufferPointer(),
    serializedRootSig->GetBufferSize(),
    IID_PPV_ARGS(mPostProcessRootSignature.GetAddressOf())));
}
```

这个根签名的定义为：根参数槽0指向一个常量缓冲区、根参数槽1指向一个SRV，根参数槽2指向一个UAV。在分派调用开始之前，我们先要为计算着色器绑定常量数据与资源描述符以供其使用：

```
cmdList->SetComputeRootSignature(rootSig);

cmdList->SetComputeRoot32BitConstants(0, 1, &blurRadius, 0);
cmdList->SetComputeRoot32BitConstants(0, (UINT)weights.size(), weights.dat
a(), 1);

cmdList->SetComputeRootDescriptorTable(1, mBlur0GpuSrv);
cmdList->SetComputeRootDescriptorTable(2, mBlur1GpuUav);

UINT numGroupsX = (UINT)ceilf(mWidth / 256.0f);
cmdList->Dispatch(numGroupsX, mHeight, 1);
```

13.3.3　利用索引对纹理进行采样

纹理元素可以借助2D索引加以访问。在13.2节定义计算着色器中，我们基于分派的线程ID来索引纹理（在13.4节中将对线程ID进行讨论）。而每个线程都要被指定一个唯一的调度ID（调度标识符）。

```
[numthreads(16, 16, 1)]
void CS(int3 dispatchThreadID : SV_DispatchThreadID)
{
  // 对两个纹理中横纵坐标分别为x、y处的纹素求和，并将结果存至相应的gOutput纹素中
  gOutput[dispatchThreadID.xy] =
    gInputA[dispatchThreadID.xy] +
    gInputB[dispatchThreadID.xy];
}
```

假设我们为处理纹理而分发了足够多的线程组（即利用一个线程来处理一个单独的纹素），那么这段代码会将两个纹理图像的对应数据进行累加，再将结果存于纹理`gOutput`中。

注意 Note

系统对计算着色器中的索引越界行为有着明确的定义。越界的读操作总是返回0，而向越界处写入数据时却不会实际执行任何操作（no-ops）[Boyd08]。

由于计算着色器运行在GPU上，因此便可以将它作为访问GPU的一般工具，特别是在通过纹理过滤来对纹理进行采样的时候。但是，这个过程中还存在两点问题。第一个问题是，我们不能使用`Sample`方法，而必须采用`SampleLevel`方法。与`Sample`相比，`SampleLevel`需要获取第三个额外的参数，以指定纹理的mipmap层级。0表示mipmap的最高级别，1是第二级，并以此类推。若此参数存在小数部分，则该小数将用于在开启mipmap线性过滤的两个mipmap层级之间进行插值。至于`Sample`方法，它会根据屏幕上纹理所覆的像素数量而自动选择最佳的mipmap层级。因为计算着色器不可直接参与渲染，它便无法知道`Sample`方法自行选择的mipmap层级，所以我们必须在计算着色器中以`SampleLevel`方法来显式（手动）指定mipmap的层级。第二个问题是，当我们对纹理进行采样时，会使用范围为$[0, 1]^2$的归一化纹理坐

标，而非整数索引。此时，我们便可以将纹理的大小（$width$，$height$）（即纹理的宽度与高度）设置为一个常量缓冲区变量，再利用整数索引(x, y)来求取归一化纹理坐标：

$$u = \frac{x}{width}$$
$$v = \frac{y}{height}$$

下列代码展示了一个使用整数索引的计算着色器，而第二个功能相同的版本则采用了纹理坐标与SampleLevel函数。这里我们假设纹理的大小为512×512，且仅使用最高的mipmap层级：

```
//
// 版本1：使用整数索引
//

cbuffer cbUpdateSettings
{
    float gWaveConstant0;
    float gWaveConstant1;
    float gWaveConstant2;

    float gDisturbMag;
    int2 gDisturbIndex;
};

RWTexture2D<float> gPrevSolInput : register(u0);
RWTexture2D<float> gCurrSolInput : register(u1);
RWTexture2D<float> gOutput       : register(u2);

[numthreads(16, 16, 1)]
void CS(int3 dispatchThreadID : SV_DispatchThreadID)
{
    int x = dispatchThreadID.x;
    int y = dispatchThreadID.y;

    gOutput[int2(x,y)] =
        gWaveConstant0 * gPrevSolInput[int2(x,y)].r +
        gWaveConstant1 * gCurrSolInput[int2(x,y)].r +
        gWaveConstant2 * (
```

```
            gCurrSolInput[int2(x,y+1)].r +
            gCurrSolInput[int2(x,y-1)].r +
            gCurrSolInput[int2(x+1,y)].r +
            gCurrSolInput[int2(x-1,y)].r);
}

//
// 版本2：使用函数SampleLevel与纹理坐标
//

cbuffer cbUpdateSettings
{
    float gWaveConstant0;
    float gWaveConstant1;
    float gWaveConstant2;

    float gDisturbMag;
    int2 gDisturbIndex;
};

SamplerState samPoint : register(s0);

RWTexture2D<float> gPrevSolInput : register(u0);
RWTexture2D<float> gCurrSolInput : register(u1);
RWTexture2D<float> gOutput    : register(u2);

[numthreads(16, 16, 1)]
void CS(int3 dispatchThreadID : SV_DispatchThreadID)
{
 // 相当于以SampleLevel()取代运算符[]
 int x = dispatchThreadID.x;
 int y = dispatchThreadID.y;

 float2 c = float2(x,y)/512.0f;
 float2 t = float2(x,y-1)/512.0;
 float2 b = float2(x,y+1)/512.0;
 float2 l = float2(x-1,y)/512.0;
 float2 r = float2(x+1,y)/512.0;

 gNextSolOutput[int2(x,y)] =
    gWaveConstants0*gPrevSolInput.SampleLevel(samPoint, c, 0.0f).r +
    gWaveConstants1*gCurrSolInput.SampleLevel(samPoint, c, 0.0f).r +
    gWaveConstants2*(
        gCurrSolInput.SampleLevel(samPoint, b, 0.0f).r +
        gCurrSolInput.SampleLevel(samPoint, t, 0.0f).r +
        gCurrSolInput.SampleLevel(samPoint, r, 0.0f).r +
        gCurrSolInput.SampleLevel(samPoint, l, 0.0f).r);
}
```

13.3.4　结构化缓冲区资源

以下示例展示了如何通过HLSL来定义结构化缓冲区（structured buffer）：

```
struct Data
{
    float3 v1;
    float2 v2;
};

StructuredBuffer<Data> gInputA : register(t0);
StructuredBuffer<Data> gInputB : register(t1);
RWStructuredBuffer<Data> gOutput : register(u0);
```

结构化缓冲区是一种由相同类型元素所构成的简单缓冲区——其本质上是一种数组。正如我们所看到的，该元素类型可以是用户以HLSL定义的结构体。

我们可以把为顶点缓冲区与索引缓冲区创建SRV的方法同样用于创建结构化缓冲区的SRV。除了“必须指定D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS标志”这一条之外，将结构化缓冲区用作UAV也与之前的操作基本一致。设置此标志的目的是用于把资源转换为D3D12_RESOURCE_STATE_UNORDERED_ACCESS状态。

```
struct Data
{
  XMFLOAT3 v1;
  XMFLOAT2 v2;
};
```

```
// 生成一些数据来填充SRV缓冲区
std::vector<Data> dataA(NumDataElements);
std::vector<Data> dataB(NumDataElements);
for(int i = 0; i < NumDataElements; ++i)
{
  dataA[i].v1 = XMFLOAT3(i, i, i);
  dataA[i].v2 = XMFLOAT2(i, 0);

  dataB[i].v1 = XMFLOAT3(-i, i, 0.0f);
  dataB[i].v2 = XMFLOAT2(0, -i);
}

UINT64 byteSize = dataA.size()*sizeof(Data);

// 创建若干缓冲区用作SRV
mInputBufferA = d3dUtil::CreateDefaultBuffer(
  md3dDevice.Get(),
  mCommandList.Get(),
  dataA.data(),
  byteSize,
  mInputUploadBufferA);

mInputBufferB = d3dUtil::CreateDefaultBuffer(
  md3dDevice.Get(),
  mCommandList.Get(),
  dataB.data(),
  byteSize,
  mInputUploadBufferB);

// 创建用作UAV的缓冲区
ThrowIfFailed(md3dDevice->CreateCommittedResource(
  &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
  D3D12_HEAP_FLAG_NONE,
  &CD3DX12_RESOURCE_DESC::Buffer(byteSize,
    D3D12_RESOURCE_FLAG_ALLOW_UNORDERED_ACCESS),
  D3D12_RESOURCE_STATE_UNORDERED_ACCESS,
  nullptr,
  IID_PPV_ARGS(&mOutputBuffer)));
```

结构化缓冲区可以像纹理那样与流水线相绑定。我们为它们创建SRV或UAV的描述符，再将这些描述符作为参数传入需要获取描述符表的根参数。或者，我们还能定义以根描述符为参数的根签名，由此便可

以将资源的虚拟地址作为根参数直接进行传递，而无须涉及描述符堆（这种方式仅限于创建缓冲区资源的SRV或UAV，并不适用于纹理）。考虑下列的根签名描述：

```
// 根参数可以是描述符表、根描述符或根常量
CD3DX12_ROOT_PARAMETER slotRootParameter[3];

// 性能优化小提示：按变更频率由高到低的顺序来填充根参数
slotRootParameter[0].InitAsShaderResourceView(0);
slotRootParameter[1].InitAsShaderResourceView(1);
slotRootParameter[2].InitAsUnorderedAccessView(0);

// 根签名由一系列根参数所构成
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(3, slotRootParameter,
    0, nullptr,
    D3D12_ROOT_SIGNATURE_FLAG_NONE);
```

接下来，我们就能绑定所创建的缓冲区以供分派调用使用：

```
mCommandList->SetComputeRootSignature(mRootSignature.Get());

mCommandList->SetComputeRootShaderResourceView(0,
  mInputBufferA->GetGPUVirtualAddress());
mCommandList->SetComputeRootShaderResourceView(1,
  mInputBufferB->GetGPUVirtualAddress());
mCommandList->SetComputeRootUnorderedAccessView(2,
  mOutputBuffer->GetGPUVirtualAddress());

mCommandList->Dispatch(1, 1, 1);
```

注意 Note

还有一种名为原始缓冲区（raw buffer）的资源，从本质上来讲，它是用字节数组来表示数据的。我们可以通过字节偏移量来找到所需数据的位置，再将它按适当的类型进行强制转换以获取数据。对于有多种

不同类型的数据存于同一缓冲区的情况来说，这种资源可谓是一股清流啊！要创建原始缓冲区资源，必须用DXGI_FORMAT_R32_TYPELESS的格式，而在创建对应的UAV时一定要使用D3D12_BUFFER_UAV_FLAG_RAW标志。本书中不会使用这种原始缓冲区资源，读者若需了解更多细节，可参考SDK文档。

13.3.5　将计算着色器的执行结果复制到系统内存

一般来说，在用计算着色器对纹理进行处理之后，我们就会将结果在屏幕上显示出来，并根据呈现的效果来验证计算着色器的准确性（accuracy）。但是，如果使用结构化缓冲区参与运算，或使用GPGPU进行通用计算，则运算结果可能根本就无法显示出来。所以当前的燃眉之急是如何将GPU端显存（您是否还记得，在通过UAV向结构化缓冲区写入数据时，缓冲区其实是位于显存之中）里的运算结果回传至系统内存中。首先，应以堆属性D3D12_HEAP_TYPE_READBACK来创建系统内存缓冲区，再通过ID3D12GraphicsCommandList::CopyResource方法将GPU资源复制到系统内存资源之中。其次，系统内存资源必须与待复制的资源有着相同的类型与大小。最后，还需用映射API函数对系统内存缓冲区进行映射，使CPU可以顺利地读取其中的数据。至此，我们就能将数据复制到系统内存块中了，可令CPU端对其开展后续的处理，或存数据于文件，或执行所需的各种操作。

本章包含了一个名为“VecAdd”的结构化缓冲区演示程序，它的功能比较简单，就是将分别存于两个结构化缓冲区中向量的对应分量进行求和运算：

```
struct Data
{
  float3 v1;
  float2 v2;
};

StructuredBuffer<Data> gInputA : register(t0);
StructuredBuffer<Data> gInputB : register(t1);
RWStructuredBuffer<Data> gOutput : register(u0);

[numthreads(32, 1, 1)]
void CS(int3 dtid : SV_DispatchThreadID)
{
  gOutput[dtid.x].v1 = gInputA[dtid.x].v1 + gInputB[dtid.x].v1;
  gOutput[dtid.x].v2 = gInputA[dtid.x].v2 + gInputB[dtid.x].v2;
}
```

为了方便起见，我们使每个结构化缓冲区中仅含有32个元素。因此，只需分派一个线程组即可（因为一个线程组即可同时处理32个数据元素）。待程序中的所有线程都完成计算着色器的运算任务之后，我们将结果复制到系统内存，再保存于文件当中。下面的代码演示了如何创建系统内存缓冲区，以及怎样将GPU中的计算结果复制到CPU的内存：

```
// 创建一个系统内存缓冲区，以便读回处理结果
ThrowIfFailed(md3dDevice->CreateCommittedResource(
  &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_READBACK),
  D3D12_HEAP_FLAG_NONE,
  &CD3DX12_RESOURCE_DESC::Buffer(byteSize),
  D3D12_RESOURCE_STATE_COPY_DEST,
  nullptr,
  IID_PPV_ARGS(&mReadBackBuffer)));

// ...
//
```

```
// 计算着色器执行完毕

struct Data
{
    XMFLOAT3 v1;
    XMFLOAT2 v2;
};

// 按计划将数据从默认缓冲区复制到回读缓冲区（即系统内存缓冲区）中
mCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
  mOutputBuffer.Get(),
  D3D12_RESOURCE_STATE_COMMON,
  D3D12_RESOURCE_STATE_COPY_SOURCE));

mCommandList->CopyResource(mReadBackBuffer.Get(), mOutputBuffer.Get());

mCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
  mOutputBuffer.Get(),
  D3D12_RESOURCE_STATE_COPY_SOURCE,
  D3D12_RESOURCE_STATE_COMMON));

// 命令记录完成
ThrowIfFailed(mCommandList->Close());

// 将命令列表添加到命令队列中用于执行
ID3D12CommandList* cmdsLists[] = { mCommandList.Get() };
mCommandQueue->ExecuteCommandLists(_countof(cmdsLists), cmdsLists);

// 等待命令执行完毕
FlushCommandQueue();

// 对数据进行映射，以便CPU读取
Data* mappedData = nullptr;
ThrowIfFailed(mReadBackBuffer->Map(0, nullptr,
   reinterpret_cast<void**>(&mappedData)));

std::ofstream fout("results.txt");

for(int i = 0; i < NumDataElements; ++i)
{
  fout << "(" << mappedData[i].v1.x << ", " <<
          mappedData[i].v1.y << ", " <<
          mappedData[i].v1.z << ", " <<
          mappedData[i].v2.x << ", " <<
          mappedData[i].v2.y << ")" << std::endl;
}

mReadBackBuffer->Unmap(0, nullptr);
```

在这个演示程序中，我们用下列初始数据来填写两个输入缓冲区：

```
std::vector<Data> dataA(NumDataElements);
std::vector<Data> dataB(NumDataElements);
for(int i = 0; i < NumDataElements; ++i)
{
  dataA[i].v1 = XMFLOAT3(i, i, i);
  dataA[i].v2 = XMFLOAT2(i, 0);

  dataB[i].v1 = XMFLOAT3(-i, i, 0.0f);
  dataB[i].v2 = XMFLOAT2(0, -i);
}
```

存有计算结果的文本文件应含有下列数据，据此我们便能确定计算着色器是否按预期完成任务：

```
(0, 0, 0, 0, 0)
(0, 2, 1, 1, -1)
(0, 4, 2, 2, -2)
(0, 6, 3, 3, -3)
(0, 8, 4, 4, -4)
(0, 10, 5, 5, -5)
(0, 12, 6, 6, -6)
(0, 14, 7, 7, -7)
(0, 16, 8, 8, -8)
(0, 18, 9, 9, -9)
(0, 20, 10, 10, -10)
(0, 22, 11, 11, -11)
(0, 24, 12, 12, -12)
(0, 26, 13, 13, -13)
(0, 28, 14, 14, -14)
(0, 30, 15, 15, -15)
(0, 32, 16, 16, -16)
(0, 34, 17, 17, -17)
(0, 36, 18, 18, -18)
(0, 38, 19, 19, -19)
(0, 40, 20, 20, -20)
(0, 42, 21, 21, -21)
(0, 44, 22, 22, -22)
(0, 46, 23, 23, -23)
(0, 48, 24, 24, -24)
(0, 50, 25, 25, -25)
```

```
(0, 52, 26, 26, -26)
(0, 54, 27, 27, -27)
(0, 56, 28, 28, -28)
(0, 58, 29, 29, -29)
(0, 60, 30, 30, -30)
(0, 62, 31, 31, -31)
```

注意 Note

观察图13.1可以发现，CPU与GPU之间的存储器复制操作最为缓慢。而对于图形处理这一角度来说，我们更是永远都不想在每一帧都执行这种复制操作，因为这样频繁地搬运数据对程序的性能而言无疑是毁灭性的的打击。有读者可能会问：在进行GPGPU编程的过程中，我们可是常常需要将运算结果返回CPU啊？然而，这对于GPGPU编程来说，往往并不是什么大难题，因为GPU运算所节省的时间远超GPU向CPU复制所花费的时间——再者说，针对GPGPU编程而言，并不是“每一帧”都要执行这种复制操作。举个例子，假设某个应用程序要通过GPGPU编程来实现一个开销极大的图形处理计算。在运算结束之后，再将其处理结果复制到CPU。在这种情况下，GPU并不会立即开始下一次处理，而是只有在用户发起另一次计算请求时，它才会为此重新开动起来。

13.4　线程标识的系统值

考虑图13.4中所示的线程分派情况。

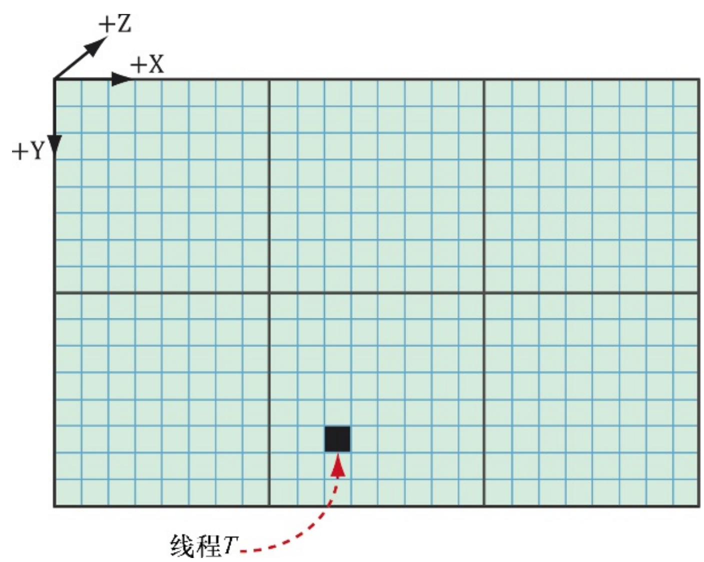


图13.4　考虑图中标出的线程T，其所在线程组的ID是 (1, 1, 0)，它在组中的线程ID为(2, 5, 0)。因此，该线程的调度线程ID为$(1,\ 1,\ 0)\otimes(8,\ 8,\ 0)+(2,\ 5,\ 0)=(10,\ 13,\ 0)$。而它在组中的线程ID则为$5\times 8+2=42$

1．系统会为每个线程组都分配一个ID，这个ID称为**线程组ID**（group ID），其系统值的语义为`SV_GroupID`。如果$G_x\times G_y\times G_z$为所分派线程组的个数，则组ID的范围为(0, 0, 0)至$(G_x-1,\ G_y-1,\ G_z-1)$。

2．在线程组中，每个线程都被指定了一个组内的唯一ID。若线程组的规模为$X\times Y\times Z$,则**组内线程ID**（group thread ID）的范围实为(0, 0, 0)到$(X-1, Y-1, Z-1)$。组内线程ID系统值的语义为`SV_GroupThreadID`。

3．调用一次Dispatch函数便会分派一个线程组网格。**调度线程ID**（dispatch thread ID，分派线程ID）是Dispatch调用为线程所生成的唯一标识（相对于所有的线程而言）。换句话说，组内线程ID是线程相对于所在线程组的唯一标识（局部），而调度线程ID则是Dispatch调用为线程指定的相对于所有线程组中全部线程的唯一识别信息（全局）。今设线程组的规模为ThreadGroupSize = (X,Y,Z)，那么我们便可以根据线程组ID与组内线程ID，通过以下方法推算出调度线程ID：

```
dispatchThreadID.xyz = groupID.xyz * ThreadGroupSize.xyz +
                       groupThreadID.xyz;
```

调度线程ID的系统值语义为SV_DispatchThreadID。如果分派了一个大小为3×2的线程组，且其中每个线程组的规模为10×10，则共分发了600个线程，而且所调度线程的ID范围为(0, 0, 0)至(29, 19, 0)。

4．通过Direct3D的系统值SV_GroupIndex便可以指定组内线程ID的线性索引，它的换算方法为：

```
groupIndex = groupThreadID.z*ThreadGroupSize.x*ThreadGroupSize.y +
             groupThreadID.y*ThreadGroupSize.x + groupThreadID.x;
```

注意　Note

至于坐标的索引顺序，其第一个坐标指出的是线程在x方向上的位置（或称“列”），第二个坐标则是线程在y方向上的位置（或称“行”）。这个顺序与普通矩阵的记法刚好相反，即M_{ij}表示矩阵中第i

行、第j列的那一个元素。

为什么需要给出这些线程ID值呢？这是因为计算着色器通常会以若干数据结构作为输入，再将计算结果输出到另一些数据结构之中。而我们就可以利用这些线程ID值来对这些数据结构进行索引：

```
Texture2D gInputA;
Texture2D gInputB;
RWTexture2D<float4> gOutput;

[numthreads(16, 16, 1)]
void CS(int3 dispatchThreadID : SV_DispatchThreadID)
{
    // 通过调度线程ID来索引输入与输出的纹理
    gOutput[dispatchThreadID.xy] =
        gInputA[dispatchThreadID.xy] +
        gInputB[dispatchThreadID.xy];
}
```

利用`SV_GroupThreadID`系统值即可极为便利地对线程的本地存储器（local storage memory）进行索引（参见13.6节）。

13.5 追加缓冲区与消费缓冲区

假设我们通过下列结构体定义了一个存有粒子数据的缓冲区：

```
struct Particle
{
    float3 Position;
    float3 Velocity;
    float3 Acceleration;
};
```

并且希望基于粒子的速度与恒定加速度在计算着色器中对其位置进行更新。此外，我们还假定不必考虑粒子的更新顺序以及它们被写入输出缓冲区的顺序。消费结构化缓冲区（consume structured buffer，一种输入缓冲区）与追加结构化缓冲区（append structured buffer，一种输出缓冲区）便是为这种场景而生的。若使用了这两种缓冲区，我们也就不必再在索引问题上花心思了：

```
struct Particle
{
    float3 Position;
    float3 Velocity;
    float3 Acceleration;
};

float TimeStep = 1.0f / 60.0f;

ConsumeStructuredBuffer<Particle> gInput;
AppendStructuredBuffer<Particle> gOutput;
[numthreads(16, 16, 1)]
void CS()
{
    // 对输入缓冲区中的数据元素之一进行处理（即“消费”，从缓冲区中移除一个元素）
    Particle p = gInput.Consume();

    p.Velocity += p.Acceleration*TimeStep;
```

```
    p.Position += p.Velocity*TimeStep;

    // 将规范化向量追加到输出缓冲区
    gOutput.Append( p );
}
```

数据元素一旦经过处理（即消费），其他线程就不能再对它进行任何操作了（事实上也就是从消费缓冲区中移除掉了）。而且，一个线程也只能处理一个数据元素。除此之外，我们无法知晓数据元素的具体处理顺序与追加顺序。因此，一般来说，某元素位于输入缓冲区的位置与其处理后写入输出缓冲区的位置（两种缓冲区中相同元素的排列顺序）并不是一一对应的。

注意 Note

追加结构化缓冲区的空间是不能动态扩展的。但是，它们一定有足够的空间来容纳我们要向其追加的所有元素。

13.6 共享内存与线程同步

每个线程组都有一块称为共享内存（shared memory）或线程本地存储器（thread local storage）的内存空间。这种内存的访问速度很快，可认为与硬件高速缓存的速度不相上下。在我们的计算着色器的代码中，共享内存的声明如下：

```
groupshared float4 gCache[256];
```

数组大小可依用户的需求而定，但是线程组共享内存的上限为32 kb。由于共享内存是线程组里的本地内存，所以要通过`SV_GroupThreadID`语义对它进行索引。据此，我们可以使组内的每个线程都来访问共享内存中的同一个元素。

使用过多的共享内存会引发性能问题[Fung10]，下面给出例子对此进行详解。假设现有一款最多支持32 KB[6]共享内存的多处理器，而用户的计算着色器则需要共享内存20 KB。这意味着只有为每个多处理器设置一个线程组才能满足此限制，因为20 KB + 20 KB = 40 KB > 32 KB，所以没有足够的共享内存供另一个线程组使用[Fung10]。这样一来就限制了GPU的并发性，因为多处理器将无法在多个线程组之间进行切换而屏蔽处理过程中的延迟（13.1节中曾提到，建议每个多处理器至少设有两个线程组）。因此，即使这款硬件在技术上仅支持32 KB的共享内存，但是通过缩减内存的使用量却能令其性能得到优化。

共享内存常见的应用场景是存储纹理数据。在特定的算法中，例如

像模糊图像（blur）这种工作，就需要对同一个纹素进行多次拾取。纹理采样实际上是一种速度较慢的GPU操作，因为内存带宽与内存延迟还未能像GPU的计算能力那样得到极大的改善[Möller08]。但是，我们可以将线程组所需的纹理样本全部预加载至共享内存块，以此来避免密集的纹理拾取操作所带来的性能下滑。接下来，算法流程便会在共享内存块中查找纹理样本并进行处理，此时的处理速度就很快了。现假设我们以下列有误的代码来实现上述方案：

```
Texture2D gInput;
RWTexture2D<float4> gOutput;

groupshared float4 gCache[256];

[numthreads(256, 1, 1)]
void CS(int3 groupThreadID : SV_GroupThreadID,
     int3 dispatchThreadID : SV_DispatchThreadID)
{
  // 每个线程都采集纹理，并将采得的数据存于共享内存中
  gCache[groupThreadID.x] = gInput[dispatchThreadID.xy];

  // 接下来执行的计算任务：访问其他线程在共享内存中存储的数据元素

  // 糟糕！！！采集左、右相邻纹素的这两条线程可能还没有完成纹理采样，并且还未将结
果存
  // 于共享内存中
  float4 left = gCache[groupThreadID.x - 1];
  float4 right = gCache[groupThreadID.x + 1];

  ...
}
```

有一个问题随之而来，根源在于我们无法保证线程组内的所有线程都能同时完成任务。这可能会导致线程访问到还未经初始化的共享内存元素，因为负责将这些元素进行初始化的相邻线程也许还没有完成它的本职工作。要填上这个坑，就一定要先等待所有的线程都将各自所处理

的纹理加载到共享内存之中，而后再令计算着色器继续后面的工作。这时就轮到同步命令闪亮登场了：

```
Texture2D gInput;
RWTexture2D<float4> gOutput;

groupshared float4 gCache[256];

[numthreads(256, 1, 1)]
void CS(int3 groupThreadID : SV_GroupThreadID,
     int3 dispatchThreadID : SV_DispatchThreadID)
{
  // 每个线程都对纹理进行采样，再将采集数据存储在共享内存中
  gCache[groupThreadID.x] = gInput[dispatchThreadID.xy];

  // 等待组内的所有线程都完成各自的任务
  GroupMemoryBarrierWithGroupSync();

  // 此时，读取共享内存的任意元素并执行计算任务都是安全的
  float4 left = gCache[groupThreadID.x - 1];
  float4 right = gCache[groupThreadID.x + 1];

  ...
}
```

13.7 图像模糊演示程序

在本节中，我们将解释如何通过计算着色器来实现令图像模糊的算法。首先叙述模糊算法的数学原理。其次，讨论渲染到纹理技术（render-to-texture），我们的例程以此来生成用于模糊处理的源图像。最后审阅计算着色器的实现代码，并探讨如何来实现那些棘手的功能细节。

13.7.1 图像模糊理论

可将图像的模糊算法描述如下：针对源图像中的每一个像素P_{ij}，计算以它为中心的$m \times n$矩阵的加权平均值（见图13.5）。此加权平均值便是经模糊处理后图像中第i行、第j列的像素颜色。用数学公式来表示即为：

$$Blur\left(P_{ij}\right) = \sum_{r=-a}^{a} \sum_{c=-b}^{b} w_{rc} P_{i+r,j+c} \text{满足} \sum_{r=-a}^{a} \sum_{c=-b}^{b} w_{rc} = 1$$

其中，$m = 2a + 1$且$n = 2b + 1$。将m与n强制为奇数，以此来保证$m \times n$矩阵总是具有“中心”项。我们称a为垂直模糊半径，b为水平模糊半径。若$a = b$，则只需指定模糊半径（blur radius）即可确定矩阵的大小。$m \times n$权值矩阵称为模糊核（blur kernel，也有译作模糊内核）。从公式中还可以看出，权值之和必为1。如果权值和小于1，则模糊后的图像将随着颜色的缺失而显得更暗；如果权值之和大于1，则模糊处理后

的图像会随着颜色的增添而更显明亮。

在保证权值和为1的前提下，我们就能用多种不同的方法来计算它。在大多数图像编辑软件中，我们能发现一种广为人知的模糊运算：高斯模糊（Gaussian blur）。此算法借助高斯函数$G(x)=\exp\left(-\frac{x^2}{2\sigma^2}\right)$来获取权值。图13.6展示了取不同$\sigma$值时高斯函数的对应图像。

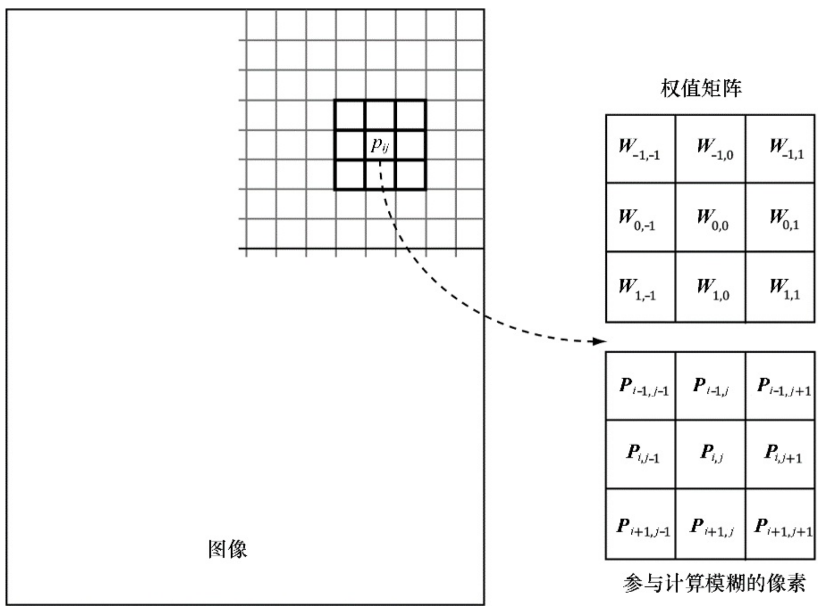


图13.5　为了对像素P_{ij}进行模糊处理，我们就要计算以此像素为中心的$m\times n$像素矩阵的加权平均值。在此图的示例中，目标矩阵是规模为3×3的方阵，模糊半径为$a=b=1$。不难看出，权值矩阵中心元素的权重w_{00}对应于像素P_{ij}

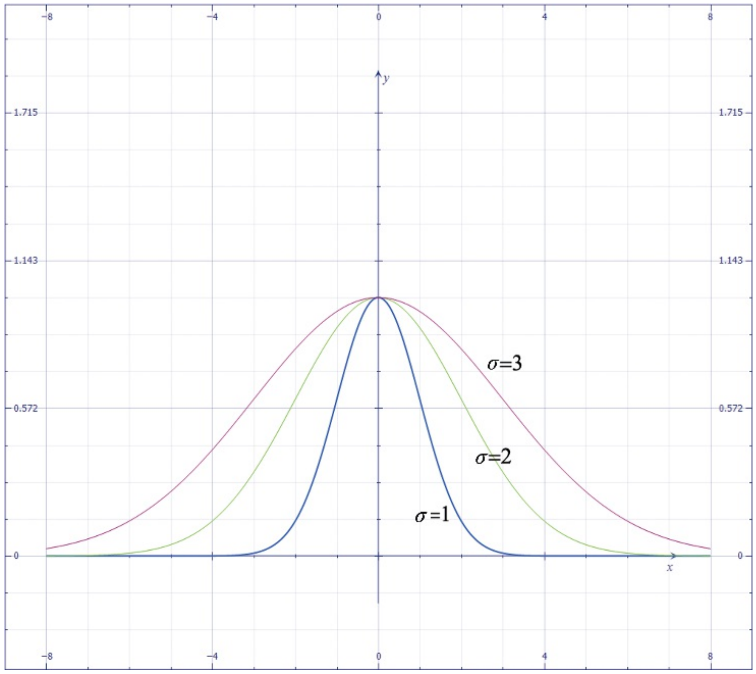


图13.6　当$\sigma = 1,\ 2,\ 3$时，函数$G(x)$的图像。可以发现，若σ越大，则曲线越趋于平缓，给邻近点所赋予的权值也就越大

现假设我们要进行规模为1×5的高斯模糊（即在水平方向上进行1D模糊），且设$\sigma = 1$。分别对$x = -2, -1, 0, 1, 2$求取$G(x)$的值，我们有：

$$G(-2) = \exp\left(-\frac{(-2)^2}{2}\right) = \mathrm{e}^{-2}$$

$$G(-1) = \exp\left(-\frac{(-1)^2}{2}\right) = \mathrm{e}^{-\frac{1}{2}}$$

$$G(0) = \exp(0) = 1$$

$$G(1) = \exp\left(-\frac{(1)^2}{2}\right) = \mathrm{e}^{-\frac{1}{2}}$$

$$G(2) = \exp\left(-\frac{(2)^2}{2}\right) = \mathrm{e}^{-2}$$

但是，这些数据还不是最终的权值，因为它们的和不为1：

$$\begin{aligned}\sum_{x=-2}^{x=2} G(x) &= G(-2)+G(-1)+G(0)+G(1)+G(2) \\ &= 1+2\mathrm{e}^{-\frac{1}{2}}+2\mathrm{e}^{-2} \\ &\approx 2.48373\end{aligned}$$

如果将上式除以和 $\sum_{x=-2}^{x=2} G(x)$ 来对它进行规范化处理，那么我们便会基于高斯函数获得总和为1的诸权值：

$$\frac{G(-2)+G(-1+G(0)+G(1)+G(2)}{\sum_{x=-2}^{x=2} G(x)} = 1$$

因此，所求高斯模糊的权重分别为：

$$w_{-2} = \frac{G(-2)}{\sum_{x=-2}^{x=2} G(x)} = \frac{\mathrm{e}^{-2}}{1+2\mathrm{e}^{-\frac{1}{2}}+2\mathrm{e}^{-2}} \approx 0.0545$$

$$w_{-1} = \frac{G(-1)}{\sum_{x=-2}^{x=2} G(x)} = \frac{\mathrm{e}^{-\frac{1}{2}}}{1+2\mathrm{e}^{-\frac{1}{2}}+2\mathrm{e}^{-2}} \approx 0.2442$$

$$w_{0} = \frac{G(0)}{\sum_{x=-2}^{x=2} G(x)} = \frac{1}{1+2\mathrm{e}^{-\frac{1}{2}}+2\mathrm{e}^{-2}} \approx 0.4026$$

$$w_{1} = \frac{G(1)}{\sum_{x=-2}^{x=2} G(x)} = \frac{\mathrm{e}^{-\frac{1}{2}}}{1+2\mathrm{e}^{-\frac{1}{2}}+2\mathrm{e}^{-2}} \approx 0.2442$$

$$w_{2} = \frac{G(2)}{\sum_{x=-2}^{x=2} G(x)} = \frac{\mathrm{e}^{-2}}{1+2\mathrm{e}^{-\frac{1}{2}}+2\mathrm{e}^{-2}} \approx 0.0545$$

高斯模糊最著名的莫过于它的可分离性（separable），根据这条性质，我们可以像下面那样将它分为两个1D模糊过程。

1．通过1D横向模糊（horizontal blur）将输入的图像I进行模糊处理：$I_H = Blur_H(I)$。

2．对上一步输出的结果再次进行1D纵向模糊（vertical blur）处理：$Blur(I) = Blur_V(I_H)$。

对公式化简一番，我们便得到：

$$Blur(I) = Blur_V(Blur_H(I))$$

假设模糊核为一个9×9矩阵，我们就需要对总计81个样本依次执行2D模糊计算。但通过将模糊过程分离为两个1D模糊阶段，便仅需处理9 + 9 = 18个样本！我们常常要对纹理进行模糊处理，而本章中也提到过：拾取纹理样本是代价高昂的操作。因此，通过分离模糊过程来减少纹理采样操作是一种受用户欢迎的优化手段。尽管有些模糊方法不具分离性（即某些模糊算子不可实现分离模糊过程），但只要保证最终图像在视觉上足够精准，我们往往还是能以优化性能为目的而简化其模糊过程。

13.7.2　渲染到纹理技术

到现在为止，我们一直都在程序中向后台缓冲区渲染数据。但是，后台缓冲区到底是怎样的一种存在呢？如果查阅了D3DApp部分的代码，我们便会发现，后台缓冲区其实就是一种位于交换链中的纹理：

```
Microsoft::WRL::ComPtr<ID3D12Resource> mSwapChainBuffer[SwapChainBufferCount];
CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHeapHandle(mRtvHeap->
```

```
    GetCPUDescriptorHandleForHeapStart());
for (UINT i = 0; i < SwapChainBufferCount; i++)
{
  ThrowIfFailed(mSwapChain->GetBuffer(i,
    IID_PPV_ARGS(&mSwapChainBuffer[i])));
  md3dDevice->CreateRenderTargetView(
    mSwapChainBuffer[i].Get(), nullptr, rtvHeapHandle);
  rtvHeapHandle.Offset(1, mRtvDescriptorSize);
}
```

通过将后台缓冲区的渲染目标视图与渲染流水线的输出合并（Output Merger，OM）阶段相绑定，使Direct3D将数据渲染至后台缓冲区中：

```
// 指定即将被渲染的缓冲区
mCommandList->OMSetRenderTargets(1, &CurrentBackBufferView(),
  true, &DepthStencilView());
```

在通过`IDXGISwapChain::Present`方法呈现后台缓冲区时，其中的数据便会显示在屏幕上。

注意　Note

用作渲染目标的纹理一定要以`D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET`标志来创建。

如果认真分析这些代码便会发现，我们还可以畅通无阻地创建另一个纹理，再为它创建渲染目标视图，并将它绑定到渲染流水线的OM阶段之上。由此，我们就能改将数据（可能还要使用不同的摄像机视角）

绘制到这种全然不同的“离屏”（off-screen）纹理之中，而非后台缓冲区之内。这种就是著名的渲染到离屏纹理（render-to-off-screen-texture）技术，简称渲染到纹理（render-to-texture）。这种纹理与后台缓冲区的唯一不同之处在于：在执行提交操作（present，亦有作呈现）的过程中，它无法显示在屏幕上。

根据以上叙述来看，渲染到纹理给人的第一感觉很可能就是“一无是处”，因为它压根就没法把纹理直接显示到屏幕上。但是，在渲染到纹理执行完毕之后，我们还可以将后台缓冲区重新绑定到OM阶段，从而继续将几何图形绘制到后台缓冲区之中。不仅如此，关键在于我们还能够用渲染到纹理期间所生成的纹理为几何体贴图。利用这个策略便可实现各种特殊的效果。例如，我们可以通过渲染到纹理技术把鸟瞰视图绘制到场景纹理之上。也就是说，在向后台缓冲区绘制数据的过程中，我们可以借此将鸟瞰图渲染到屏幕右下角的一个小方框中，以此来模拟雷达系统（见图13.7）。渲染到纹理的用武之地还有：

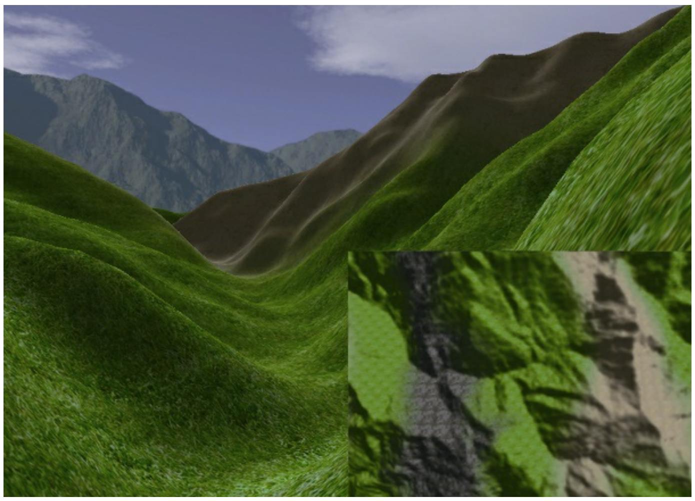


图13.7　利用一个位于玩家上方的摄像机以鸟瞰图的视角将场景渲染至一个离屏纹理之中。在把场景以玩家视角绘至后台缓冲区的时候，我们将上述离屏纹理渲染至屏幕右下角的方框之

中，以此来显示雷达图

1．阴影贴图（shadow mapping）。

2．屏幕空间环境光遮蔽（screen space ambient occlusion，SSAO）。

3．动态反射与立方体图（dynamic reflections with cube maps）。

若使用渲染到纹理技术，则在GPU上实现的模糊算法将以如下方式工作：把演示程序中的场景按寻常方式渲染到离屏纹理上，该纹理会被输入至计算着色器并执行模糊算法。待纹理经模糊处理后，我们会将所得的纹理绘制为全屏四边形（full screen quad）并送到后台缓冲区，由此便可根据模糊效果来检验模糊的实现。此流程的关键步骤可概括如下：

1．像往常一样将场景绘制到一个离屏纹理之中。

2．通过计算着色器程序来对该纹理进行模糊处理。

3．将后台缓冲区恢复为渲染目标，并以模糊后的纹理来绘制全屏四边形。

使用渲染到纹理技术来实现模糊效果比较妥帖，而且在需要将场景渲染为与后台缓冲区大小不同的纹理中时，这种方法也是理想的选择。假设离屏纹理与后台缓冲区的大小及格式相匹配，我们就能采用间接绘制到离屏纹理的方式代之：像之前一样先将纹理渲染至后台缓冲区，再

把后台缓冲区的内容用CopyResource方法复制到离屏纹理。接下来，我们就能在离屏纹理上开展计算工作，再将模糊后的纹理绘制为全屏四边形送至后台缓冲区，以生成最终的屏幕输出。

```
// 将input（在这个示例中它是后台缓冲区资源）复制到BlurMap0。
cmdList->CopyResource(mBlurMap0.Get(), input);
```

以上就是实现模糊演示程序所用到的技术。在练习6中，我们将通过渲染到纹理技术完成另一种与之截然不同的过滤器。

注意 Note

上述处理过程需要先用普通的渲染流水线进行绘制，继而切换到计算着色器执行计算任务，最后再切换回普通的渲染流水线。一般来讲，我们应当尝试避免在渲染与计算工作之间的往复切换行为，因为上下文（context）的切换会产生开销[NVIDIA10]。在每一帧中应试着先完成所有的计算工作，再执行全部的渲染任务。当然，这有时是无法实现的。例如，在上述处理过程中，我们需要先把场景渲染到一个纹理，使它在计算着色器中进行模糊处理，然后将处理的结果绘制出来。虽然不能完全避免切换操作，但是我们还可以尝试将切换的次数降到最低。

13.7.3 图像模糊的实现概述

首先，假设所用模糊算法具有可分离性，据此将模糊操作分为两个1D模糊运算—— 一个横向模糊运算，一个纵向模糊运算。实现这种算法需要两个可读写的纹理缓冲区，也就是说，需要为两个纹理分别创建SRV与UAV。从现在开始，我们称这两个纹理为纹理**A**与纹理**B**。据此，则模糊算法的处理过程如下：

1．给纹理**A**绑定SRV，作为计算着色器的输入（我们会对此输入图像进行横向模糊处理）。

2．给纹理**B**绑定UAV，作为计算着色器的输出（该输出图像存有横向模糊计算后的数据）。

3．分派线程组执行横向模糊操作。完成后，纹理**B**会存储横向模糊的结果$\text{Blur}_H(I)$，这里的I就是接受模糊处理的输入图像。

4．为纹理**B**绑定SRV，作为计算着色器的输入（此图像就是即将进行纵向模糊且已执行过横向模糊的图像）。

5．为纹理**A**绑定UAV，作为计算着色器的输出（该输出纹理会存有最终的模糊图像数据）。

6．分发线程组来执行纵向模糊操作。待处理完毕后，纹理**A**会保存最后的模糊结果$\text{Blur}(I)$，其中，I即是最原始的输入图像。

这一系列逻辑实现了具有可分离性质模糊公式$\text{Blur}(I) = \text{Blur}_V(\text{Blur}_H(I))$的图像处理工作。可以看出，纹理**A**与纹理**B**在某些时刻分别充当了计算着色器的输入与输出，但是无法同时担任两种

角色（在将一个资源同时绑定为着色器的输入与输出之时，Direct3D便会报错）。将横向模糊过程（blur pass）与纵向模糊过程结合起来就能组成一个完整的模糊过程。对处理后的图像再次进行模糊处理，便可以使它变得愈加模糊。我们可以对图像反复进行模糊处理，直至达到满意的效果。

由于渲染到纹理中的场景与窗口工作区要保持着相同的分辨率，因此我们需要不时重新构建离屏纹理，而模糊算法所用的第二个纹理**B**的缓冲区也是如此。这可以通过`OnResize`方法来实现：

```
void BlurApp::OnResize()
{
  D3DApp::OnResize();

  // 窗口大小有了变化，所以要更新纵横比，并重新计算投影矩阵
  XMMATRIX P = XMMatrixPerspectiveFovLH(
    0.25f*MathHelper::Pi, AspectRatio(),
    1.0f, 1000.0f);
  XMStoreFloat4x4(&mProj, P);

  if(mBlurFilter != nullptr)
  {
    mBlurFilter->OnResize(mClientWidth, mClientHeight);
  }
}

void BlurFilter::OnResize(UINT newWidth, UINT newHeight)
{
  if((mWidth != newWidth) || (mHeight != newHeight))
  {
    mWidth = newWidth;
    mHeight = newHeight;

    // 以新的大小来重新构建离屏纹理资源
    BuildResources();

    // 既然创建了新的资源，我们也应当为其创建新的描述符
    BuildDescriptors();
  }
```

```
}
```

变量mBlurFilter是我们所编写的BlurFilter辅助类实例。此类不仅封装了纹理**A**与纹理**B**的SRV、UAV和纹理资源，还提供了开启计算着色器中实际模糊运算的方法。我们即将讨论此辅助类的具体实现。

上面曾提到BlurFilter类封装了纹理资源。为了能够使用绘制/分派命令，就应当将这些资源绑定到流水线上，因此，便需要为这些资源创建相应的描述符。同时，这也就意味着我们一定要在D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV类型的描述符堆中开辟额外的空间来存储这些描述符。BlurFilter::BuildDescriptors方法就是利用在堆中处于起始位置的描述符句柄来存储BlurFilter类要用到的描述符。该方法缓存了模糊过程中所需的一切描述符句柄，并以此来创建相应的描述符。利用该函数缓存句柄的原因在于，当窗口的大小发生改变时，它可以随资源的变化而重新创建这些描述符：

```
void BlurFilter::BuildDescriptors(
  CD3DX12_CPU_DESCRIPTOR_HANDLE hCpuDescriptor,
  CD3DX12_GPU_DESCRIPTOR_HANDLE hGpuDescriptor,
  UINT descriptorSize)
{
  // 保存对描述符的引用
  mBlur0CpuSrv = hCpuDescriptor;
  mBlur0CpuUav = hCpuDescriptor.Offset(1, descriptorSize);
  mBlur1CpuSrv = hCpuDescriptor.Offset(1, descriptorSize);
  mBlur1CpuUav = hCpuDescriptor.Offset(1, descriptorSize);

  mBlur0GpuSrv = hGpuDescriptor;
  mBlur0GpuUav = hGpuDescriptor.Offset(1, descriptorSize);
  mBlur1GpuSrv = hGpuDescriptor.Offset(1, descriptorSize);
  mBlur1GpuUav = hGpuDescriptor.Offset(1, descriptorSize);

  BuildDescriptors();
}
```

```
void BlurFilter::BuildDescriptors()
{
  D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
  srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPI
NG;
  srvDesc.Format = mFormat;
  srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
  srvDesc.Texture2D.MostDetailedMip = 0;
  srvDesc.Texture2D.MipLevels = 1;

  D3D12_UNORDERED_ACCESS_VIEW_DESC uavDesc = {};

  uavDesc.Format = mFormat;
  uavDesc.ViewDimension = D3D12_UAV_DIMENSION_TEXTURE2D;
  uavDesc.Texture2D.MipSlice = 0;

  md3dDevice->CreateShaderResourceView(mBlurMap0.Get(), &srvDesc,
    mBlur0CpuSrv);
  md3dDevice->CreateUnorderedAccessView(mBlurMap0.Get(),
    nullptr, &uavDesc, mBlur0CpuUav);

  md3dDevice->CreateShaderResourceView(mBlurMap1.Get(), &srvDesc,
    mBlur1CpuSrv);
  md3dDevice->CreateUnorderedAccessView(mBlurMap1.Get(),
    nullptr, &uavDesc, mBlur1CpuUav);
}

// 以下代码位于BlurApp.cpp文件中...
// 创建BlurFilter类所用资源的描述符来填充描述符堆
mBlurFilter->BuildDescriptors(
  CD3DX12_CPU_DESCRIPTOR_HANDLE(
    mCbvSrvUavDescriptorHeap->GetCPUDescriptorHandleForHeapStart(),
    3, mCbvSrvUavDescriptorSize),
  CD3DX12_GPU_DESCRIPTOR_HANDLE(
    mCbvSrvUavDescriptorHeap->GetGPUDescriptorHandleForHeapStart(),
    3, mCbvSrvUavDescriptorSize),
  mCbvSrvUavDescriptorSize);
```

注意 Note

对图像进行模糊处理是一种昂贵的操作，它所花费的时间与待处理

的图像大小息息相关。一般情况下，在把场景渲染到离屏纹理的时候，我们通常会将离屏纹理的大小设为后台缓冲区尺寸的1/4。也就是说，假使后台缓冲区的大小为800×600，则离屏纹理的尺寸将为400×300。这样一来不仅能加快离屏纹理的绘制速度（即减少了需要填充的像素数量），而且能同时提升模糊图像的处理速度（需要模糊的像素也变得更少）。另外，当纹理从1/4的屏幕分辨率拉伸为完整的屏幕分辨率时，纹理放大（magnification）过滤器也会执行一些额外的模糊操作。

假设要处理的图像宽为w、高为h。正如我们将在下一小节中所看到的计算着色器代码所写，对于1D横向模糊而言，一个线程组用256个线程来处理水平方向上的线段，而且每个线程又负责图像中一个像素的模糊操作。因此，为了图像中的每个像素都能得到模糊处理，我们需要在x方向上分派$ceil\left(\frac{w}{256}\right)$个线程组（$ceil$为向上取整函数），且在$y$方向上调度$h$个线程组。如果$w$不能被256整除，则最后一次分派的线程组会存有多余的线程（见图13.8）。我们对于这种情况无能为力，因为线程组的大小固定。因此，我们只得把注意力放在着色器代码中越界问题的钳位检测（clamping check）之上。

1D纵向模糊与上述1D横向模糊的情况相似。在纵向模糊过程中，线程组就像由256个线程构成的垂直线段，每个线程只负责图像中一个像素的模糊运算。因此，为了使图像中的每个像素都能得到模糊处理，我们需要在y方向上分派$ceil\left(\frac{h}{256}\right)$个线程组，并在$x$方向上调度$w$个线程

组。

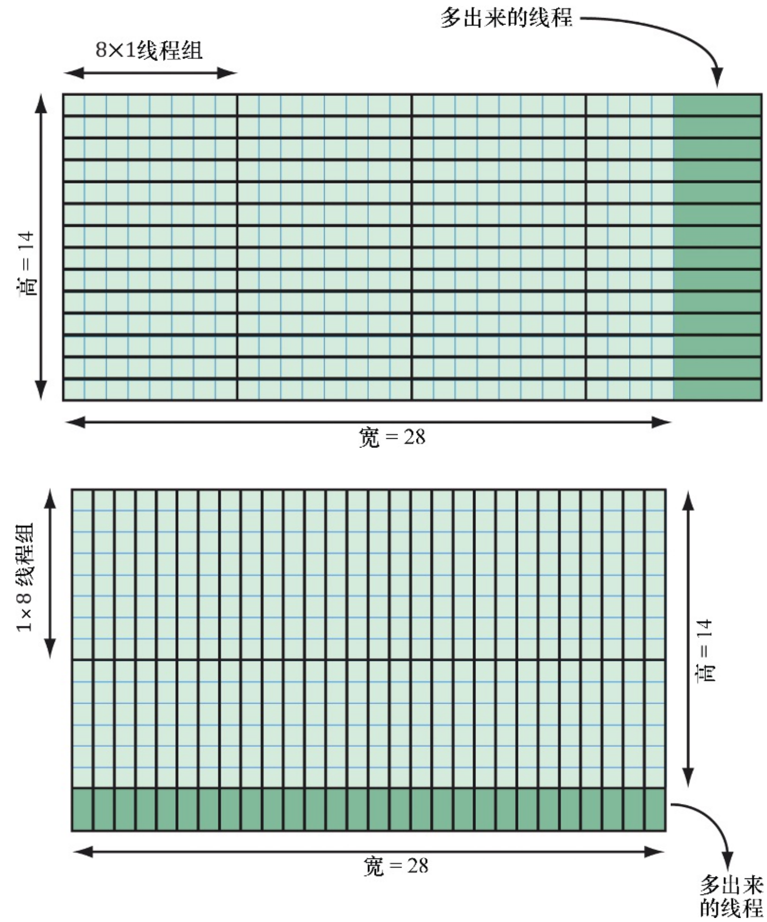


图13.8　现在来考虑对一个28×14像素的纹理进行处理，我们所用的横向、纵向线程组的规模分别为8×1与1×8（采用的是$X \times Y$的表示格式）。对于水平方向的处理过程来说，为了处理所有的像素，我们需要在x方向上分派$ceil\left(\frac{w}{8}\right) = ceil\left(\frac{28}{8}\right) = 4$个线程组，并在$y$方向上调度14个线程组。由于28并非8的整数倍，所以最右侧的线程组中会有$(4 \times 8 - 28) \times 14 = 56$个线程什么都不做。对于垂直方向的处理过程而言，为了处理所有的像素，我们需要在y方向上分派$ceil\left(\frac{h}{8}\right) = ceil\left(\frac{14}{8}\right) = 2$个线程组，并在$x$方向上调度28个线程组。同理，由于14并不是8的整数倍，所以最下侧的线程组中会有$(2 \times 8 - 14) \times 28$个闲置的线程。沿用这同一思路即可将线程组拓展为256个线程的规模来处理更大的纹理

下列代码不仅计算出了每个方向上要分派的线程组数量，还真正地

开启了计算着色器的模糊运算：

```
void BlurFilter::Execute(ID3D12GraphicsCommandList* cmdList,
                ID3D12RootSignature* rootSig,
                ID3D12PipelineState* horzBlurPSO,
                ID3D12PipelineState* vertBlurPSO,
                ID3D12Resource* input,
                int blurCount)
{
  auto weights = CalcGaussWeights(2.5f);
  int blurRadius = (int)weights.size() / 2;

  cmdList->SetComputeRootSignature(rootSig);

  cmdList->SetComputeRoot32BitConstants(0, 1, &blurRadius, 0);
  cmdList->SetComputeRoot32BitConstants(0, (UINT)weights.size(),
     weights.data(), 1);

  cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(input,
    D3D12_RESOURCE_STATE_RENDER_TARGET, D3D12_RESOURCE_STATE_COPY_SOURCE))
;

  cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
   mBlurMap0.Get(),
   D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_COPY_DEST));

  // 将输入的资源数据（此例中是后台缓冲区）复制到BlurMap0
  cmdList->CopyResource(mBlurMap0.Get(), input);

  cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
    mBlurMap0.Get(),
    D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_GENERIC_READ));

  cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
    mBlurMap1.Get(),
    D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_UNORDERED_ACCESS));

for(int i = 0; i < blurCount; ++i)
{
  //
  // 水平方向上的模糊处理过程
  //

  cmdList->SetPipelineState(horzBlurPSO);

  cmdList->SetComputeRootDescriptorTable(1, mBlur0GpuSrv);
```

```
  cmdList->SetComputeRootDescriptorTable(2, mBlur1GpuUav);

  // 若每个线程组能处理256个像素（256这个值是在计算着色器中定义的），那么处理一
行像素需要分派
  // 几个线程组呢
  UINT numGroupsX = (UINT)ceilf(mWidth / 256.0f);
  cmdList->Dispatch(numGroupsX, mHeight, 1);

  cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
    mBlurMap0.Get(),
    D3D12_RESOURCE_STATE_GENERIC_READ,
    D3D12_RESOURCE_STATE_UNORDERED_ACCESS));

  cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
    mBlurMap1.Get(),
    D3D12_RESOURCE_STATE_UNORDERED_ACCESS,
    D3D12_RESOURCE_STATE_GENERIC_READ));

  //
  // 垂直方向上的模糊处理过程
  //

  cmdList->SetPipelineState(vertBlurPSO);

  cmdList->SetComputeRootDescriptorTable(1, mBlur1GpuSrv);
  cmdList->SetComputeRootDescriptorTable(2, mBlur0GpuUav);

  // 每个线程组能处理256个像素（256这个值是在计算着色器中定义的），那么需要分派
几个线程组才能
  // 处理一列像素呢
  UINT numGroupsY = (UINT)ceilf(mHeight / 256.0f);
  cmdList->Dispatch(mWidth, numGroupsY, 1);

  cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
    mBlurMap0.Get(),
    D3D12_RESOURCE_STATE_UNORDERED_ACCESS,
    D3D12_RESOURCE_STATE_GENERIC_READ));

  cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
    mBlurMap1.Get(),
    D3D12_RESOURCE_STATE_GENERIC_READ,
    D3D12_RESOURCE_STATE_UNORDERED_ACCESS));
  }
}
```

图13.9所示为“Blur”（模糊）演示程序的效果。

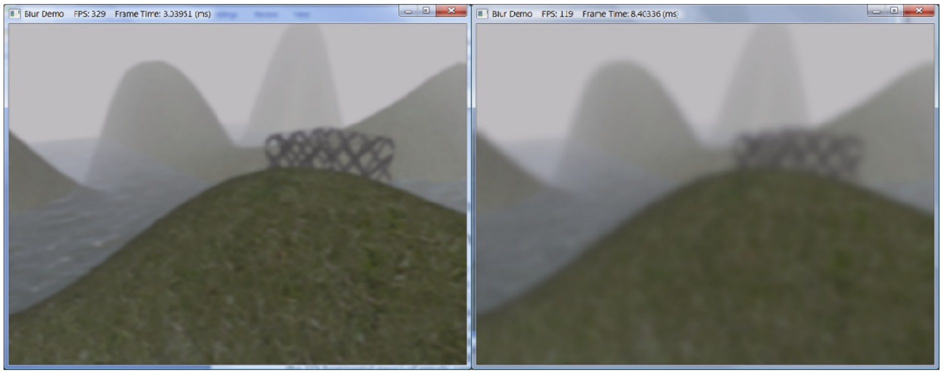


图13.9　左图，“Blur”（模糊）演示程序的效果，此图像为经两次模糊处理后的效果
右图，同是“Blur”演示程序的效果，但达到此效果要经8次模糊处理

13.7.4　计算着色器程序

在这一节中，我们将查阅实际执行模糊运算的计算着色器程序。不过，我们仅讨论水平模糊这一种情况，因为垂直模糊的细节与之相仿，只是方向上不同而已。

就像上一小节中所提到的，我们分派的线程组是由256个线程构成的水平“线段”，每个线程都负责图像中一个像素的模糊操作。首先要讲解的是按部就班实现模糊算法的低效方案，即每个线程都简单地计算出以正在处理的像素为中心的行矩阵（因为我们正在进行的是1D横向模糊处理，所以要针对行矩阵进行计算）的加权平均值。这个方法的缺点是需要多次拾取同一纹素（见图13.10）。

我们可以根据13.6节中所述的模糊处理策略，利用共享内存来优化上述算法。这样一来，每个线程就可以在共享内存中读取或存储其所需的纹素数据。待所有线程都从共享内存读取它们所需的纹素之后，就能

够执行模糊运算了。不得不说，从共享内存中读取数据的速度飞快。除此之外，还有一件棘手的事，即利用具有n= 256个线程的线程组行模糊运算的时候，却需要共$n+2R$个纹素数据，这里的R就是模糊半径（见图13.11）。

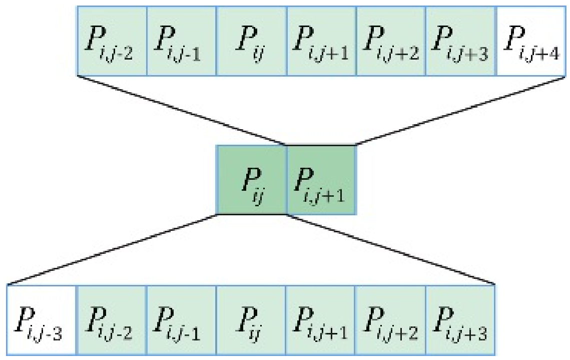


图13.10　仅考虑输入图像中的这两个相邻像素，假设模糊核为1×7。不难看出，在对这两个像素进行模糊的过程中，8个像素中有6个被采集了2次，即每个模糊像素过程各对这6个像素采集了1遍



图13.11　由于模糊半径的原因，在处理线程组边界附近的像素时，可能会读取线程组以外存在"越界"情况的像素

解决办法其实也并不复杂。我们只需分配出能容下$n+2R$个元素的共享内存，并且有$2R$个线程要各获取两个纹素数据。唯一麻烦的地方就是在索引共享内存时要多花些心思，因为组内线程ID此时不能与共享内存中的元素一一对应了。图13.12演示了当$R=4$时，从线程到共享内存的映射过程。

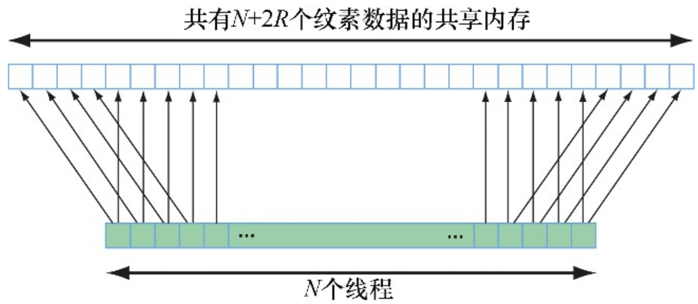


图13.12　在此例中，$R=4$。最左侧的4个线程以及最右侧的4个线程，每个都要读取2个纹素数据，并将它们存于共享内存之中。而这8个线程之外的所有线程只需读取1个纹素，并将其存于共享内存

之中。这样一来，我们即可得到以模糊半径R对N个像素进行模糊处理所需的所有纹素数据

现在要讨论的是最后一种情况，即图13.13中所示的最左侧与最右侧的线程组在索引输入图像时会发生越界的情形。

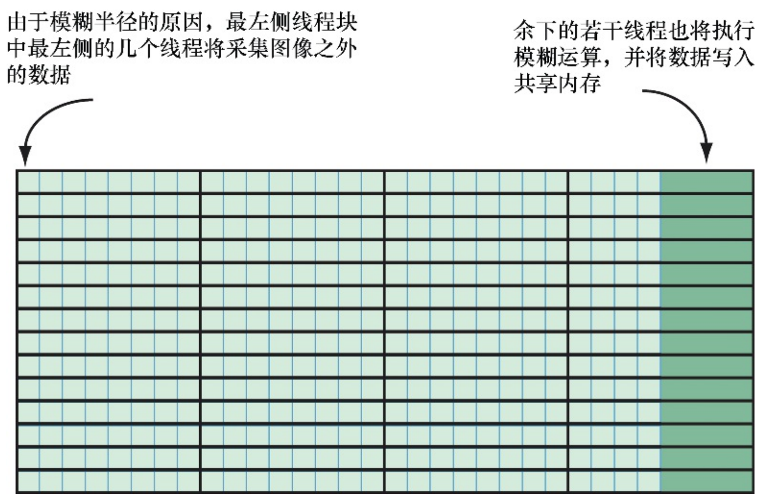


图13.13　我们可能对图像边界以外的数据进行读取的情况

从越界的索引处读取数据并不是非法操作——与之对应的行为定义是返回0（对越界索引处进行写入是不会执行任何操作的，即no-op）。然而，我们在读取越界数据时却不希望得到数据0，因为这意味着值为0的颜色（即黑色）会影响到边界处的模糊结果。我们此时期盼能实现出

类似于钳位（clamp）纹理寻址模式的效果，即在读取越界的数据时，能够获得一个与边界纹素相同的数据。这个方案可通过对索引进行钳位来加以实现：

```
// 针对图像左侧边界处的越界采样情况进行钳位操作
int x = max(dispatchThreadID.x - gBlurRadius, 0);
gCache[groupThreadID.x] = gInput[int2(x, dispatchThreadID.y)];

// 对于图像右侧边界处所存在的越界采样情况进行的钳位操作
int x = min(dispatchThreadID.x + gBlurRadius, gInput.Length.x-1);
gCache[groupThreadID.x+2*gBlurRadius] = gInput[int2(x, dispatchThreadID.y)
];

// 对图像边界处存在的越界采样情况进行钳位操作
gCache[groupThreadID.x+gBlurRadius] =
gInput[min(dispatchThreadID.xy, gInput.Length.xy-1)];
```

完整的着色器代码如下。

```
// =====================================================================
// 计算模糊半径最多为5个像素的可分离高斯模糊
// =====================================================================

cbuffer cbSettings : register(b0)
{
  // 我们不能把根常量映射到位于常量缓冲区中的数组元素，因此要将每一个元素都一一列
出

  int gBlurRadius;

  // 最多支持11个模糊权值
  float w0;
  float w1;
  float w2;
  float w3;
  float w4;
  float w5;
  float w6;
  float w7;
  float w8;
  float w9;
  float w10;
```

```
};

static const int gMaxBlurRadius = 5;

Texture2D gInput             : register(t0);
RWTexture2D<float4> gOutput : register(u0);

#define N 256
#define CacheSize (N + 2*gMaxBlurRadius)
groupshared float4 gCache[CacheSize];

[numthreads(N, 1, 1)]
void HorzBlurCS(int3 groupThreadID : SV_GroupThreadID,
                      int3 dispatchThreadID : SV_DispatchThreadID)
{
  // 放在数组中便于索引
  float weights[11] = { w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10 };

  //
  // 通过填写本地线程存储区来减少带宽的负载。若要对N个像素进行模糊处理，根据模糊
半径，我们需要
  // 加载N + 2*BlurRadius个像素
  //

  // 此线程组运行着N个线程。为了获取额外的2*BlurRadius个像素，就需要有2*BlurRad
ius个线程
  // 都多采集一个像素数据
  if(groupThreadID.x < gBlurRadius)
  {
    // 对于图像左侧边界存在越界采样的情况进行钳位操作
    int x = max(dispatchThreadID.x - gBlurRadius, 0);
    gCache[groupThreadID.x] = gInput[int2(x, dispatchThreadID.y)];
  }
  if(groupThreadID.x >= N-gBlurRadius)
  {
    // 对于图像右侧边界处的越界采样情况进行钳位操作
    int x = min(dispatchThreadID.x + gBlurRadius, gInput.Length.x-1);
    gCache[groupThreadID.x+2*gBlurRadius] = gInput[int2(x, dispatchThreadI
D.y)];
  }

  // 针对图像边界处的越界采样情况进行钳位处理
  gCache[groupThreadID.x+gBlurRadius] = gInput[min(dispatchThreadID.xy,
    gInput.Length.xy-1)];

  // 等待所有的线程完成任务
  GroupMemoryBarrierWithGroupSync();
```

```
  //
  // 现在对每个像素进行模糊处理
  //

  float4 blurColor = float4(0, 0, 0, 0);

  for(int i = -gBlurRadius; i <= gBlurRadius; ++i)
  {
    int k = groupThreadID.x + gBlurRadius + i;

    blurColor += weights[i+gBlurRadius]*gCache[k];
  }

  gOutput[dispatchThreadID.xy] = blurColor;
}

[numthreads(1, N, 1)]
void VertBlurCS(int3 groupThreadID : SV_GroupThreadID,
                int3 dispatchThreadID : SV_DispatchThreadID)
{
  // 放入数组中便于索引
  float weights[11] = { w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10 };

  //
  // 填写本地线程存储器来减少带宽的负荷。如果要对N个像素进行模糊处理，再加上模糊
半径，我们就需
  // 要加载共N + 2*BlurRadius个像素
  //

  // 该线程组运行着N个线程。要取得另外2*BlurRadius个像素的话，就需要有2*BlurRad
ius个线程
  // 要额外多采集一个像素
  if(groupThreadID.y < gBlurRadius)
  {
    // 对于图像上侧边界处的越界采样情况进行钳位处理
    int y = max(dispatchThreadID.y - gBlurRadius, 0);
    gCache[groupThreadID.y] = gInput[int2(dispatchThreadID.x, y)];
  }
  if(groupThreadID.y >= N-gBlurRadius)
  {
    // 针对图像下侧边界处的越界采样情况进行钳位处理
    int y = min(dispatchThreadID.y + gBlurRadius, gInput.Length.y-1);
    gCache[groupThreadID.y+2*gBlurRadius] = gInput[int2(dispatchThreadID.x
, y)];
  }

  // 对于图像边界处的越界采样情况进行钳位处理
  gCache[groupThreadID.y+gBlurRadius] = gInput[min(dispatchThreadID.xy,
```

```
    gInput.Length.xy-1)];

  // 等待所有的线程都完成各自的任务
  GroupMemoryBarrierWithGroupSync();

  //
  // 现在对每一个像素都进行模糊处理
  //

  float4 blurColor = float4(0, 0, 0, 0);

  for(int i = -gBlurRadius; i <= gBlurRadius; ++i)
  {
    int k = groupThreadID.y + gBlurRadius + i;

    blurColor += weights[i+gBlurRadius]*gCache[k];
  }

  gOutput[dispatchThreadID.xy] = blurColor;
}
```

至于最后一行代码

```
gOutput[dispatchThreadID.xy] = blurColor;
```

最右侧的线程组可能存有一些多余的线程，但输出的纹理中并没有与之对应的元素（意即它们根本无需输出任何数据，见图13.13）。此时，`dispatchThreadID.xy`即为输出纹理之外的一个越界索引。但是我们无须为此而担心，因为向越界处写入数据的效果是不进行任何操作（no-op）。

13.8　拓展资料

计算着色器编程自成一体，有几本书专门讲解GPU在通用计算程序方面上的应用。

1．《Programming Massively Parallel Processors: A Hands-on Approach》（《大规模并行处理器编程实战》《大规模并行处理器程序设计》），作者为David B. Kirk和 Wen-mei W. Hwu。

2．《OpenCL Programming Guide》（《OpenCL编程指南》），作者为Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fung和Dan Ginsburg。

CUDA与OpenCL其实就是以访问GPU来编写通用计算程序的两组不同的API。对CUDA与OpenCL程序的最佳实践也就是对DirectCompute[7]编程的最佳实践，因为这几类功能类似的程序都运行在相同的硬件之上。在本章中，我们已经接触了DirectCompute编程中的大部分语法，由此，我们可以轻而易举地将CUDA与OpenCL程序向DirectCompute移植。

Chuck Walbourn发布的一篇博客上含有一些介绍DirectCompute技术的链接：

http://blogs.msdn.com/b/chuckw/archive/2010/07/14/directcompute.aspx

另外，微软公司的Channel 9有一系列关于DirectCompute的讲解视频《DirectCompute Lecture Series》：

http://channel9.msdn.com/tags/DirectCompute-Lecture-Series/

最后要提到的是，NVIDIA公司还针对CUDA技术专门设置了一整套培训课程《Existing University Courses》。

尤其是那些出自伊利诺伊大学的完整CUDA编程视频讲座，我们在此极力推荐。再次重申，我们可以把CUDA理解为用于访问GPU计算功能的另一种API。只要理解了这些语法，我们就掌握了编写高效GPU通用计算程序的关键部分。通过学习这些CUDA讲座，我们将对GPU硬件的工作机制有更加深入的理解，继而写出优化程度更高的代码。

13.9　小结

1．调用`ID3D12GraphicsCommandList::Dispatch`这个API即可分派一个线程组网格。每个线程组都是一个由线程构成的3D网格，线程组中的线程数由计算着色器里的`[numthreads(x,y,z)]`属性来指定。出于对性能的考量，线程的总数应为warp大小（这是NVIDIA公司所生产硬件的基本调度单位，它的大小是32）的整数倍或wavefront尺寸（这是AMD（ATI）公司所生产硬件的基本调度单位，其大小为64）的整数倍。

2．为了保证处理的并行性（parallelism），应至少为每个多处理器分派两个线程组。所以，若硬件具有16个多处理器，则应当至少调度32个线程组，以确保所有的多处理器每时每刻都在工作。未来的硬件很可能会载有更多的多处理器，所以我们编写的程序也应适当地调高线程组数量，以便针对未来的硬件设备进行扩展。

3．一旦把线程组分配给多处理器，NVIDIA硬件就会将组中的线程按32个一组划分为warp。而后，多处理器会以SIMD的方式（即同一warp中的每个线程都执行相同的指令）调度以多个线程所构成的warp进行处理工作。如果一个warp因处理抓取纹理内存这样的工作而暂时停止运行，则多处理器会迅速切换到另一个warp并执行此warp中的相应指令，以此来屏蔽这种暂停的情况。这会使多处理器连续不停地运转，以保持忙碌的状态，从而充分发挥其计算能力。现在就能解释为什么我们建议将线程组的大小设置为warp尺寸的整数倍了。若非如此，则在线程

组向warp划分的过程中，会有warp被掺入什么都不做的无用线程。

4．若要通过计算着色器访问纹理资源，可为输入的纹理创建SRV，再将其与计算着色器相绑定。RWTexture2D是一种可供计算着色器读写的纹理，我们可以通过为纹理创建UAV（无序访问视图）并把它与计算着色器相绑定来创建这种纹理。纹理元素可用[]运算符表示法进行索引，或者通过纹理坐标、SamplerState（采样器状态）配合SampleLevel方法来开展采样。

5．结构化缓冲区是一种由相同类型元素构成的缓冲区，这与数组有些相似。其元素类型可以是用户自定义的结构体。只读结构化缓冲区的HLSL定义是这样的：

```
StructuredBuffer<DataType> gInputA;
```

而用HLSL定义可读写结构化缓冲区的方法为：

```
RWStructuredBuffer<DataType> gOutput;
```

要让计算着色器访问只读缓冲区资源，只需为它创建对应的SRV，再将其绑定到计算着色器上即可；若要计算着色器访问可读写缓冲区资源，仅需为其创建可供读写的UAV，而后再将它与计算着色器相绑定。

6．各种类型的线程ID可通过系统值传入计算着色器中。这些ID则通常会用作资源与共享内存的索引。

7．消费结构化缓冲区与追加结构化缓冲区在HLSL中的定义如下：

```
ConsumeStructuredBuffer<DataType> gInput;
AppendStructuredBuffer<DataType> gOutput;
```

如果数据元素的处理顺序与最终写入输出缓冲区中的顺序是无关紧要的，那么这两种结构缓冲区将是不错的选择，因为它们能使我们绕开繁琐的索引语法。要注意的是，追加缓冲区的空间并不能自动按需增长，但是它们一定有足够空间来容下我们向其追加的所有数据元素。

8．所有的线程组都有一块被称为共享内存或线程本地存储器的空间。该共享内存的访问速度极快，可以与硬件缓存比肩。而且，此共享内存对于性能优化或实现特定算法极为有益。在计算着色器的代码中，共享内存的声明如下：

```
groupshared float4 gCache[N];
```

N是用户所需的数组大小，但是要注意，线程组共享内存的上限是32kb。假设一个多处理器最多支持32 kb的共享内存，考虑到性能，一个线程组所用的共享内存应不多于16 kb，否则一个多处理器将无法交替运行两个这样的线程组，从而难以保证其持续运转。

9．尽量避免在计算处理与渲染过程之间进行切换，因为这会产生开销。一般来讲，我们在每一帧中应先尝试完成所有的计算任务，而后再执行后续的所有渲染工作。

13.10 练习

1．编写一个计算着色器，令其输入为具有64个3D向量（向量的模为范围[1, 10]内的随机数）的结构化缓冲区。此计算着色器的功能是计算向量的长度，并将结果输出到一个浮点缓冲区之中。最后，把运算结果复制到CPU端的内存之中，再转存至文件内。程序执行后，要验证所有向量的长度是否在范围[1, 10]之间。

2．用特定类型的缓冲区再次实现练习 1。即以`Buffer<float3>`定义输入缓冲区，再令`Buffer<float>`作为输出缓冲区。

3．假设以上习题中的向量都进行规范化处理，而且它们在缓冲区中的排列顺序是无关紧要的，试利用追加缓冲区与消费缓冲区再次实现练习1。

4．研究双边模糊（bilateral blur，也称作双边滤波器，Bilateral filter）技术，并用计算着色器来加以实现。最后，以此技术来完成另一版本的“Blur”（模糊）演示程序。

5．在此之前，我们已在演示程序里通过*Waves.h/.cpp*文件中的`Waves`类，利用CPU来计算2D波浪的模拟运动方程。现将此功能交由GPU端处理，并以类型为`float`的纹理来分别存储之前帧、当前帧以及后续帧的浪高数据。由于UAV是可读写资源，所以仅使用此类型的视图即可，而无须再涉及SRV：

```
  RWTexture2D<float> gPrevSolInput : register(u0);
  RWTexture2D<float> gCurrSolInput : register(u1);
  RWTexture2D<float> gOutput       : register(u2);
```

下面就可以利用计算着色器来执行波浪的更新运算了。运用一个独立的计算着色器生成水波，可以使这一工作过程免受其他任务的干扰。在完成栅格高度的更新后，我们就可以利用与其顶点有着同样分辨率的波浪纹理来渲染三角形栅格（所以每个栅格顶点都有与之对应的纹素），再将当前的波浪纹理绑定到以下的新“波浪”顶点着色器。接下来，我们就能在顶点着色器中对纹理进行采样并将其移动到相应的位置高度上（这称为位移贴图，displacement mapping，也有译作置换贴图、位移映射等），再估算出法线。

```
VertexOut VS(VertexIn vin)
{
  VertexOut vout = (VertexOut)0.0f;


#ifdef DISPLACEMENT_MAP
  // 使用未经变换的纹理坐标在范围[0,1]^2内采集位移贴图
  vin.PosL.y += gDisplacementMap.SampleLevel(gsamLinearWrap, vin.TexC, 1.0
f).r;
  // 通过有限差分（finite difference）来估算法线
  float du = gDisplacementMapTexelSize.x;
  float dv = gDisplacementMapTexelSize.y;
  float l = gDisplacementMap.SampleLevel( gsamPointClamp,
    vin.TexC-float2(du, 0.0f), 0.0f ).r;
  float r = gDisplacementMap.SampleLevel( gsamPointClamp,
    vin.TexC+float2(du, 0.0f), 0.0f ).r;
  float t = gDisplacementMap.SampleLevel( gsamPointClamp,
    vin.TexC-float2(0.0f, dv), 0.0f ).r;
  float b = gDisplacementMap.SampleLevel( gsamPointClamp,
    vin.TexC+float2(0.0f, dv), 0.0f ).r;

  vin.NormalL = normalize( float3(-r+l, 2.0f*gGridSpatialStep, b-t) );

#endif
```

```
  // 把顶点变换到世界空间
  float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
  vout.PosW = posW.xyz;

  // 假设这里要执行的是等比缩放，否则需使用世界矩阵的逆转置矩阵
  vout.NormalW = mul(vin.NormalL, (float3x3)gWorld);

  // 将顶点转换到齐次裁剪空间
  vout.PosH = mul(posW, gViewProj);

  // 为三角形插值而输出顶点属性
  float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
  vout.TexC = mul(texC, gMatTransform).xy;

  return vout;
}
```

最后，以512×512的栅格点为例，将发布模式（release mode）下的GPU实现与CPU实现进行性能的比对。

6. 索贝尔算子（Sobel Operator）用于图像的边缘检测。它会针对每一个像素估算其梯度（gradient）的大小。有着较大梯度的像素即表明它与周围像素的颜色差异极大，因而此像素一定位于图像的边缘。相反，具有较小梯度的像素则意味着它与临近像素的颜色趋于相同，也就是说，该像素并不处于图像边沿之上。注意，索贝尔算子返回的并非是像素是否位于图像边缘的二元结果，而是一个范围在[0, 1]内表示边沿“陡峭”程度的灰度值：值为0表示平坦，即该像素不处于图像边缘（该像素与周围像素并没有颜色差异）；值为1则表示非常陡峭，即该像素处于边沿或图像不连续（此像素与其周围像素的颜色差异较大）。索贝尔逆图像$(1-c)$往往会更加直观有效，这时白色表示平坦且不位于图像边缘，而黑色则代表陡峭且处于图像边沿（见图13.14）。

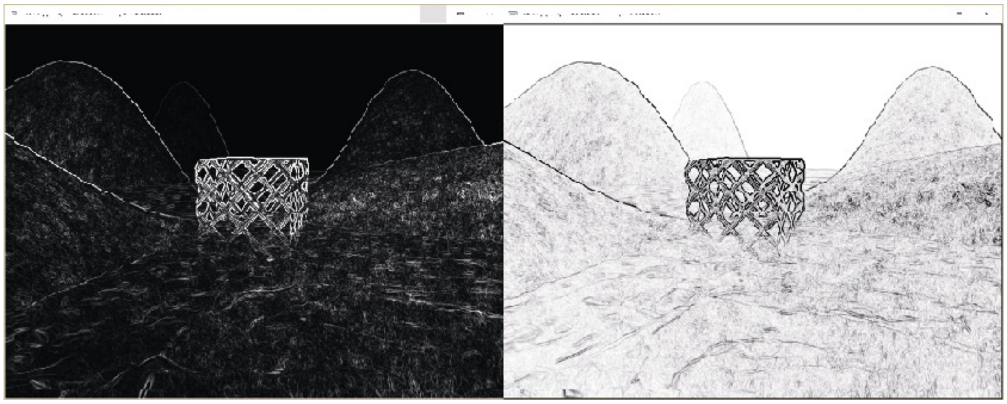


图13.14　（左图）运用索贝尔算子之后的图像效果，白色像素表示图像边缘。（右图）在索贝尔算子的逆图像中，则用黑色像素表示图像边缘

如果将原始图像与其经过索贝尔算子生成的逆图像两者间的对应颜色值相乘，我们将获得类似于卡通画或动漫书中那样，其边缘就像用黑色的笔勾描后的图片效果（见图13.15）。我们可以用这种算子实现上述效果，哪怕待处理的图像首先经过模糊处理后已隐去了部分细节，依旧可恢复其相对粗犷的画风，令其边缘明晰起来。处理过程是：利用索贝尔算子构建出模糊图像的边缘检测图像，再将此勾勒出边缘的索贝尔逆图像与原模糊图像进行乘法运算。

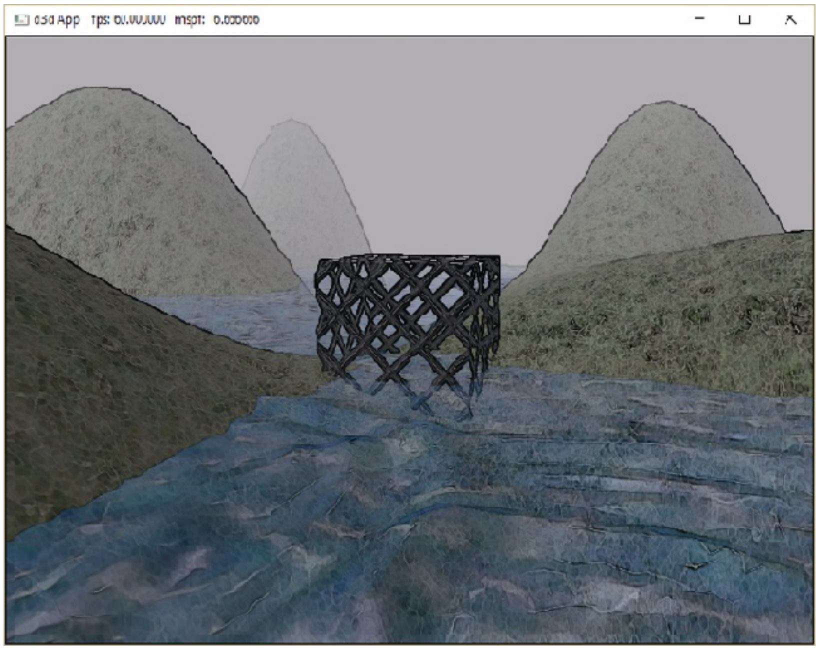


图13.15　将原始图像与边缘检测的逆图像相乘所产生的如黑笔描绘图像边缘的效果

现使用渲染到纹理技术与计算着色器来实现索贝尔算子。在借助索贝尔算子生成边缘检测图像之后，将其逆图像与原始图像相乘，以此来得到图13.15中所示的效果。着色器部分需要含有下列代码[8]。

```
//=============================================================================
// 通过索贝尔算子来进行图像的边缘检测
//=============================================================================

Texture2D gInput      : register(t0);
RWTexture2D<float4> gOutput : register(u0);

// 根据RGB 数据计算出对应亮度(luminance，即“亮度(brightness)”) 的近似值。这些权重是在人
// 眼对不同光的波长敏感度的实验基础之上所得来的
float CalcLuminance(float3 color)
{
  return dot(color, float3(0.299f, 0.587f, 0.114f));
}

[numthreads(16, 16, 1)]
void SobelCS(int3 dispatchThreadID : SV_DispatchThreadID)
{
  // 采集与当前欲处理像素相邻的众像素
```

```
  float4 c[3][3];
  for(int i = 0; i < 3; ++i)
  {
    for(int j = 0; j < 3; ++j)
    {
      int2 xy = dispatchThreadID.xy + int2(-1 + j, -1 + i);
      c[i][j] = gInput[xy];
    }
  }

  // 针对每个颜色通道，运用索贝尔公式估算出关于x的偏导数近似值
  float4 Gx = -1.0f*c[0][0] - 2.0f*c[1][0] - 1.0f*c[2][0] +
    1.0f*c[0][2] + 2.0f*c[1][2] + 1.0f*c[2][2];

  // 对于每个颜色通道，利用索贝尔公式估算出关于y的偏导数近似值
  float4 Gy = -1.0f*c[2][0] - 2.0f*c[2][1] - 1.0f*c[2][2] +
    1.0f*c[0][0] + 2.0f*c[0][1] + 1.0f*c[0][2];

  // 梯度即为(Gx, Gy)。针对每个颜色通道，计算出梯度的大小（梯度的模）以找到最大
的变化率
  float4 mag = sqrt(Gx*Gx + Gy*Gy);

  // 将梯度陡峭的边缘处绘制为黑色，梯度平坦的非边缘处绘制为白色
  mag = 1.0f - saturate(CalcLuminance(mag.rgb));

  gOutput[dispatchThreadID.xy] = mag;
}

//**********************************************************************
****
// Composite.hlsl的作者为Frank Luna (C) 2015版权所有
//
// 将两张图像组合在一起
//**********************************************************************
****

Texture2D gBaseMap : register(t0);
Texture2D gEdgeMap : register(t1);

SamplerState gsamPointWrap     : register(s0);
SamplerState gsamPointClamp    : register(s1);
SamplerState gsamLinearWrap    : register(s2);
SamplerState gsamLinearClamp   : register(s3);
SamplerState gsamAnisotropicWrap : register(s4);
SamplerState gsamAnisotropicClamp : register(s5);

static const float2 gTexCoords[6] =
{
```

```
  float2(0.0f, 1.0f),
  float2(0.0f, 0.0f),
  float2(1.0f, 0.0f),
  float2(0.0f, 1.0f),
  float2(1.0f, 0.0f),
  float2(1.0f, 1.0f)
};

struct VertexOut
{
  float4 PosH  : SV_POSITION;
  float2 TexC  : TEXCOORD;
};

VertexOut VS(uint vid : SV_VertexID)
{
  VertexOut vout;

  vout.TexC = gTexCoords[vid];

  // 将[0,1]^2区间映射到NDC（规格化设备坐标）空间
  vout.PosH = float4(2.0f*vout.TexC.x - 1.0f, 1.0f - 2.0f*vout.TexC.y, 0.0
f,
   1.0f);

  return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
  float4 c = gBaseMap.SampleLevel(gsamPointClamp, pin.TexC, 0.0f);
  float4 e = gEdgeMap.SampleLevel(gsamPointClamp, pin.TexC, 0.0f);

  // 将边缘图与原始图像相乘
  return c*e;
}
```

---

[1] 文中这段讲述的都是GPU的内部结构，请勿将核芯显卡（集成显卡以及部分独立显卡）所共享的系统内存与之混淆。

[2] 组线程数量与分派函数参数的设置配合很有讲究，直接关乎计算着色器的工作效率！

[3]　已被AMD公司收购。

[4]　将渲染（图形）流水线与计算流水线视作两种不同的流水线。另外，D3D12_COMPUTE_PIPELINE_STATE_DESC结构体共有5个字段，由于没有涉及其他两个的有关技术（如多GPU），所以并未提及。

[5]　前面谈到了渲染（图形）流水线与计算流水线的区别，那么对应的绘制请求与线程组分派也就分别被称之为“绘制调用（draw call）”与“分派调用（dispatch call，调度调用）”。

[6]　微软文档上写D3D10与D3D11中支持组内共享内存分别为16kb与32kb，本书原文也记作kb。NVIDIA文档上上却写D3D10与D3D11中支持组内共享内存分别为16KB与32KB。而2010年AMD Radeon HD 2000系列的文档中也提出，shader model 4.0（对应DirectX 10）与Shader model 5.0（对应DirectX 11）分别支持组内共享内存为16KB与32KB。尽管通过文档下文（及惯用法）可以推断KB表示的是kilobyte（千字节），但是粗看还是容易产生歧义性。

[7]　本章所讨论的计算着色器（compute shader）又称DirectCompute技术（有时写作Direct Compute）。在DirectX 10中仅支持部分功能子集（DirectCompute 4.0），于DirectX 11中正式初次发表完整功能版本（DirectCompute 5.0）。

[8]　简单来说，梯度从本质上来讲也是一种向量，借助其长度（即模）就可以反映出（像素值间的）变化程度，从而确定当前像素是否处于变化剧烈的图像分割边缘。索贝尔算子与高斯模糊的算法有些相似，也是

根据样本周围的像素值进行加权处理，只不过根据功能不同，因而权重取值不同，且计算的是梯度有关数据。书中忽略了其推导过程与具体算法细节描述，直接给出了代码，欲知详情可查询有关文献。译者认为顶点着色器VS中将y轴从[0,1]区间向NDC空间变换的代码存疑，如果按照其代码所示，y轴坐标将被变换至区间[1-2*0,1-2*1]，即区间[1,-1]。

第14章　曲面细分阶段

曲面细分阶段[1]（tessellation stage，直译作镶嵌阶段或镶嵌化处理阶段）是指渲染流水线中参与对几何图形进行镶嵌处理（tessellating geometry）的3个阶段。简而言之，曲面细分技术就是将几何体细分为更小的三角形，并以某种方式把这些新生成的顶点偏移到合适的位置，从而以增加三角形数量的方式丰富网格的细节。但是，为什么不在创建网格之初就直接赋予它高模（high-poly，高多边形）的细节呢？以下是使用曲面细分的3个理由。

1．基于GPU实现动态LOD（Level of Detail，细节级别）。可以根据网格与摄像机的距离或依据其他因素来调整其细节。比如说，若网格离摄像机较远，则按高模的规格对它进行渲染将是一种浪费，因为在那个距离我们根本看不清网格所有细节。随着物体与摄像机之间距离的拉近，我们就能连续地对它镶嵌细分，以增加物体的细节。

2．物理模拟与动画特效。我们可以在低模（low-poly，也有译作低面多边形等）网格上执行物理模拟与动画特效的相关计算，再以镶嵌化处理手段来获取细节更加丰富的网格。这种降低物理模拟与动画特效计算量的做法能够节省不少的计算资源。

3．节约内存。我们可以在各种存储器（磁盘、RAM与VRAM）中保存低模网格，再根据需求用GPU动态地对网格进行镶嵌细分。

图14.1展示了位于顶点着色器与几何着色器之间的曲面细分阶，但本书在此章之前从未用到过它们，由此可见，这3个阶段都是可选的。

学习目标：

1．了解曲面细分所用的面片图元类型。

2．理解曲面细分阶段中的每个步骤都做了什么，它们所需的输入及输出又分别是哪种数据。

3．通过编写外壳着色器与域着色器程序来对几何图形进行镶嵌化细分。

4．熟悉不同的细分策略，以便在镶嵌化处理时选择出最适当的方案。除此之外，还要知晓硬件曲面细分的性能。

5．学习贝塞尔曲线与贝塞尔曲面的数学描述，并在曲面细分阶段将它们予以实现。

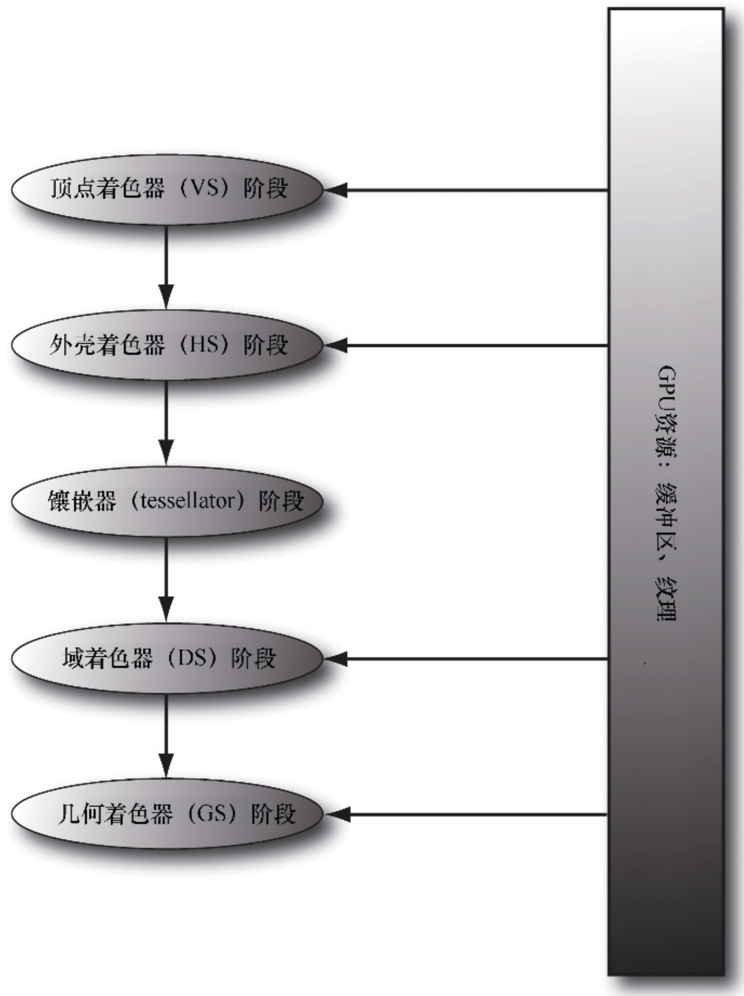


图14.1　图中展示的渲染流水线子集便是曲面细分阶段（即处于中间位置的3个渲染步骤）

14.1　曲面细分的图元类型

在进行曲面细分时，我们并不向IA（输入装配）阶段提交三角形，而是提交具有若干控制点（control point）的面片（patch）。Direct3D支持具有1～32个控制点的面片，并以下列图元类型进行描述。

```
D3D_PRIMITIVE_TOPOLOGY_1_CONTROL_POINT_PATCHLIST = 33,
D3D_PRIMITIVE_TOPOLOGY_2_CONTROL_POINT_PATCHLIST = 34,
D3D_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST = 35,
D3D_PRIMITIVE_TOPOLOGY_4_CONTROL_POINT_PATCHLIST = 36,
.
.
.
D3D_PRIMITIVE_TOPOLOGY_31_CONTROL_POINT_PATCHLIST = 63,
D3D_PRIMITIVE_TOPOLOGY_32_CONTROL_POINT_PATCHLIST = 64,
```

由于可以将三角形看作是拥有3个控制点的三角形面片（`D3D_PRIMITIVE_3_CONTROL_POINT_PATCH`），所以我们依然可以提交需要镶嵌化处理的普通三角形网格。对于简单的四边形面片而言，则只需提交具有4个控制点的面片（`D3D_PRIMITIVE_4_CONTROL_POINT_PATCH`）即可。这些面片最终也会在曲面细分阶段经镶嵌化处理而分解为多个三角形[2]。

注意 Note

在向`ID3D12GraphicsCommandList::IASetPrimitiveTopology`方法传递控制点图元类型时，我们还需

将D3D12_GRAPHICS_PIPELINE_STATE_DESC::Primitive-TopologyType字段设置为D3D12_PRIMITIVE_TOPOLOGY_TYPE_PATCH，如：opaque-PsoDesc.PrimitiveTopologyType=D3D12_PRIMITIVE_TOPOLOGY_TY

那么，具有更多控制点的面片又有什么用处呢？控制点的概念来自于特定种类数学曲线或数学曲面的构造过程。如果在类似于Adobe Illustrator这样的绘图程序中使用过贝塞尔曲线工具，那读者一定会知道要通过控制点才能描绘出曲线形状。在数学上，可以利用贝塞尔曲线来生成贝塞尔曲面。举个例子，我们可以用9个控制点或16个控制点来创建一个贝塞尔四边形面片，所用的控制点越多，我们对面片形状的控制也就越随心所欲。因此，这一切图元控制类型都是为了给这些不同种类的曲线、曲面的绘制提供支持。我们会在本章给出贝塞尔四边形面片的示例以及相关解释。

曲面细分与顶点着色器

在我们向渲染流水线提交了面片的控制点后，它们就会被推送至顶点着色器。这样一来，在开启曲面细分之时，顶点着色器就彻底沦陷为“处理控制点的着色器”。正因如此，我们还能在曲面细分开展之前，对控制点进行一些调整。一般来讲，动画与物理模拟的计算工作都会在对几何体进行镶嵌化处理之前的顶点着色器中以较低的频次进行（镶嵌

化处理后，顶点增多，处理的频次也将随之增加）。

14.2　外壳着色器

在以下小节中，我们会探索外壳着色器（hull shader），它实际上是由两种着色器（phase）组成的：

1．常量外壳着色器。

2．控制点外壳着色器。

14.2.1　常量外壳着色器

常量外壳着色器（constant hull shader）会针对每个面片逐一进行处理（即每处理一个面片就被调用一次），它的任务是输出网格的曲面细分因子（tessellation factor，也有译作细分因子、镶嵌因子等）。曲面细分因子指示了在曲面细分阶段中将面片镶嵌处理后的份数。下面是一个具有4个控制点的四边形面片（quad patch）示例，我们将它从各个方面均匀地镶嵌细分为3份。

```
struct PatchTess
{
  float EdgeTess[4]   : SV_TessFactor;
  float InsideTess[2] : SV_InsideTessFactor;

  // 可以在下面为每个面片附加所需的额外信息
};

PatchTess ConstantHS(InputPatch<VertexOut, 4> patch,
           uint patchID : SV_PrimitiveID)
{
  PatchTess pt;
```

```
    // 将该面片从各方面均匀地镶嵌处理为3等份

    pt.EdgeTess[0] = 3; // 四边形面片的左侧边缘
    pt.EdgeTess[1] = 3; // 四边形面片的上侧边缘
    pt.EdgeTess[2] = 3; // 四边形面片的右侧边缘
    pt.EdgeTess[3] = 3; // 四边形面片的下侧边缘

    pt.InsideTess[0] = 3; // u轴（四边形内部细分的列数）
    pt.InsideTess[1] = 3; // v轴（四边形内部细分的行数）

    return pt;
}
```

常量外壳着色器以面片的所有控制点作为输入，在此用InputPatch<VertexOut, 4>对此进行定义。前面提到，控制点首先会传至顶点着色器，因此它们的类型由顶点着色器的输出类型VertexOut来确定。在此例中，我们的面片拥有4个控制点，所以就将InputPatch模板的第二个参数指定为4。系统还通过SV_PrimitiveID语义提供了面片的ID值，此ID唯一地标识了绘制调用过程中的各个面片，我们可根据具体的需求来运用它。常量外壳着色器必须输出曲面细分因子，该因子取决于面片的拓扑结构。

注意 Note

除了曲面细分因子（SV_TessFactor与SV_InsideTessFactor，分别表示几何图形边缘与内部的细分份数）之外，我们还能令常量外壳着色器输出其他的面片信息。域着色器接收来自常量外壳着色器的输出数据作为输入，继而使用这些额外的面片信息。

对四边形面片进行镶嵌化处理的过程由两个部分构成：

1．4个边缘曲面细分因子控制着对应边缘镶嵌后的份数。

2．两个内部曲面细分因子指示了如何来对该四边形面片的内部进行镶嵌化处理（一个曲面细分因子针对四边形的横向维度，另一个则作用于四边形的纵向维度）。

在14.3节中，图14.2所示的例子展示了使用不同的曲面细分因子会生成结构各异的四边形面片。届时我们将研究这几组示例，直到对边缘与内部这两种曲面细分因子的工作原理了如指掌。

对三角形面片（triangle patch）执行镶嵌化处理的过程同样分为两部分：

1．3个边缘曲面细分因子控制着对应边上镶嵌后的份数。

2．一个内部曲面细分因子指示着三角形面片内部的镶嵌份数。

在14.3节中，图14.3所示的例子即通过不同的曲面细分因子最终得到的结构各异的三角形面片。

Direct3D 11硬件所支持的最大曲面细分因子为64[3]。如果把所有的曲面细分因子都设置为0，则该面片会被后续的处理阶段丢弃。这就使我们能够以每个面片为基准来实现如视锥体剔除（frustum culling）与背面剔除这类优化。

1．如果面片根本没有出现在视锥体范围内，那么就能将它从后续的处理中丢弃（倘若已经对该面片进行了镶嵌化处理，那么其细分后的各三角形将在三角形裁剪（triangle clipping）期间被抛弃）。

2．如果面片是背面朝向的，那么就能将其从后面的处理过程中丢弃（如果该面片已经过了镶嵌化处理，则其细分后的诸三角形会在光栅化阶段的背面剔除过程中被遗弃）。

一个问题自然而然地浮现出来：到底应该执行几次镶嵌化处理才合适？前面提到，曲面细分的基本想法就是为了丰富网格的细节。但是，如果用户对此无感，我们就不必无谓地为它增添细节了。以下是一些确定镶嵌次数的常用衡量标准。

1．根据与摄像机之间的距离：物体与摄像机的距离越远，能分辨的细节就越少。因此，我们在两者距离较远时渲染物体的低模版本，并随着两者逐渐接近而逐步对物体进行更加细致的镶嵌化细分。

2．根据占用屏幕的范围：可以先估算出物体覆盖屏幕的像素个数。如果数量比较少，则渲染物体的低模版本。随着物体占用屏幕范围的增加，我们便可以逐渐增大镶嵌化细分因子。

3．根据三角形的朝向：三角形相对于观察者的朝向也被列入考虑的范畴之中。位于物体轮廓边缘（silhouette edge）[4]上的三角形势必比其他位置的三角形拥有更多的细节。

4．根据粗糙程度：粗糙不平的表面较光滑的表面需要进行更为细

致的曲面细分处理。通过对表面纹理进行检测可以预算出相应的粗糙度数据，继而来决定镶嵌化处理的次数。

[Story10]给出了以下几点关于性能的建议。

1．如果曲面细分因子为1（这个数值其实意味着该面片不必细分），那么就考虑在渲染此面片时不对它进行细分处理；否则，便会在曲面细分阶段白白浪费GPU资源，因为在此阶段并不对其执行任何操作。

2．考虑到性能又涉及GPU对曲面细分的具体实现，所以不要对小于8个像素这种过小的三角形进行镶嵌化处理。

3．使用曲面细分技术时要采用批绘制调用（batch draw call，即尽量将曲线细分任务集中执行）（在绘制调用之间往复开启、关闭曲面细分功能的代价极其高昂）。

14.2.2　控制点外壳着色器

控制点外壳着色器（control point hull shader）以大量的控制点作为输入与输出，每输出一个控制点，此着色器都会被调用一次。该外壳着色器的应用之一是改变曲面的表示方式，比如说把一个普通的三角形（向渲染流水线提交的3个控制点）转换为3次贝塞尔三角形面片（cubic Bézier triangle patch，即一种具有10个控制点的面片）。例如，假设我们像平常那样利用（具有3个控制点的）三角形对网格进行建模，就可

以通过控制点外壳着色器，将这些三角形转换为具有10个控制点的高阶三次贝塞尔三角形面片。新增的控制点不仅会带来更丰富的细节，而且能将三角形面片镶嵌细分为用户所期望的份数。这一策略被称为**N-patches**方法（法线—面片方法，normal-patches scheme）或**PN**三角形方法（即（曲面）点—法线三角形方法，（curved） point-normal triangles，简记作PN triangles scheme）[Vlachos01]。由于这种方案只需用曲面细分技术来改进已存在的三角形网格，且无须改动美术制作流程，所以实现起来比较方便。对于本章第一个演示程序来说，控制点外壳着色器仅充当一个简单的传递着色器（pass-through shader），它不会对控制点进行任何的修改。

注意 Note

驱动程序可能会对传递着色器进行检测与优化[Bilodeau10b]。

```
struct HullOut
{
  float3 PosL : POSITION;
};

[domain("quad")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(4)]
[patchconstantfunc("ConstantHS")]
[maxtessfactor(64.0f)]
HullOut HS(InputPatch<VertexOut, 4> p,
      uint i : SV_OutputControlPointID,
      uint patchId : SV_PrimitiveID)
```

```
{
  HullOut hout;

  hout.PosL = p[i].PosL;

  return hout;
}
```

通过InputPatch参数即可将面片的所有控制点都传至外壳着色器之中。系统值SV_OutputControlPointID索引的是正在被外壳着色器所处理的输出控制点。值得注意的是，输入的控制点数量与输出的控制点数量未必相同。例如，输入的面片可能仅含有4个控制点，而输出的面片却能够拥有16个控制点；意即，这些多出来的控制点可由输入的4个控制点所衍生。

上面的控制点外壳着色器还用到了以下几种属性。

1．domain：面片的类型。可选用的参数有tri（三角形面片）、quad（四边形面片）或isoline（等值线）。

2．partitioning：指定了曲面细分的细分模式。

a．integer。新顶点的添加或移除仅取决于曲面细分因子的整数部分，而忽略它的小数部分。这样一来，在网格随着曲面细分级别而改变时，会容易发生明显的突跃（popping）的情况[5]。

b．非整型曲面细分（fractional_even/fractional_odd）。新顶点的添加或移除取决于曲面细分因子的整数部分，但是细微的渐变“过渡”调整就要根据因子的小数部分。当我们希望将粗糙的网格经曲

面细分而平滑地过渡到具有更佳细节的网格时，该参数就派上用场了。理解整型细分与非整型细分之间差别的最佳方式就是通过动画实际演示、比对，因此，本章末会有相应的练习来令读者比较两者的差异。

3．outputtopology：通过细分所创的三角形的绕序[6]。

a．triangle_cw：顺时针方向的绕序。

b．triangle_ccw：逆时针方向的绕序。

c．line：针对线段曲面细分。

4．outputcontrolpoints：外壳着色器执行的次数，每次执行都输出1个控制点。系统值SV_OutputControlPointID给出的索引标明了当前正在工作的外壳着色器所输出的控制点。

5．patchconstantfunc：指定常量外壳着色器函数名称的字符串。

6．maxtessfactor：告知驱动程序，用户在着色器中所用的曲面细分因子的最大值。如果硬件知道了此上限，便可了解曲面细分所需的资源，继而就能在后台对此进行优化。Direct3D 11硬件支持的曲面细分因子最大值为64。

14.3　镶嵌器阶段

程序员无法对镶嵌器[7]这一阶段进行任何控制，因为这一步骤的操作全权交由硬件处理。此环节会基于常量外壳着色器程序所输出的曲面细分因子，对面片进行镶嵌化处理。图14.2和图14.3详细展示了根据不同的曲面细分因子，对四边形面片与三角形面片所进行的不同细分操作。

14.3.1　四边形面片的曲面细分示例

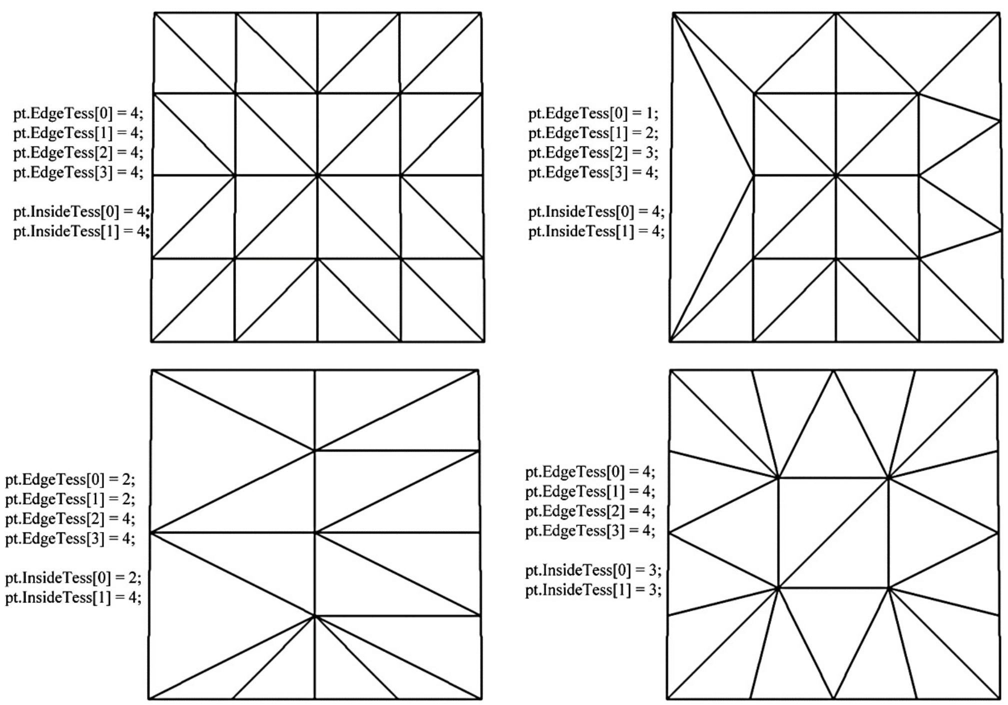


图14.2　基于不同的边缘细分因子以及内部细分因子对四边形细分的示意图

14.3.2　三角形面片的曲面细分示例

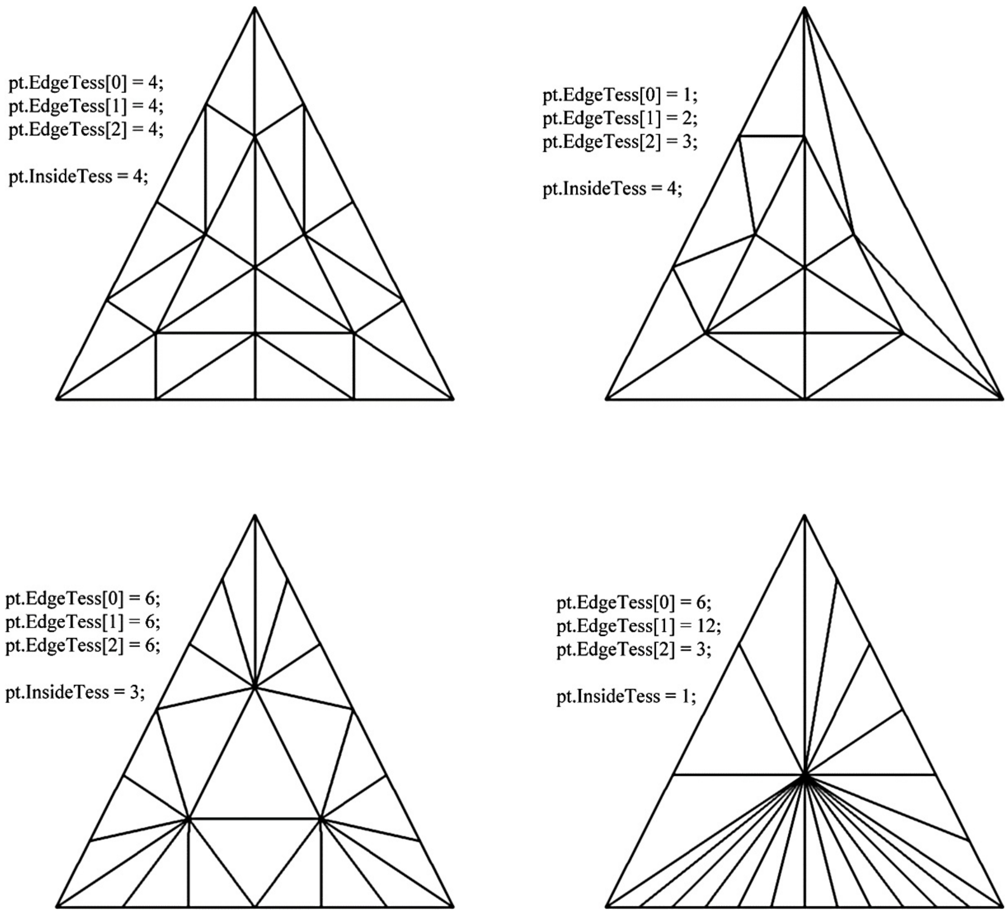


图14.3　基于不同的边缘细分因子以及内部细分因子对三角形细分的示意图

14.4　域着色器

镶嵌器阶段会输出新建的所有顶点与三角形，在此阶段所创建顶点，都会逐一调用域着色器（domain shader）进行后续处理。随着曲面细分功能的开启，顶点着色器便化身为“处理每个控制点的顶点着色器”，而外壳着色器[8]的本质实为“针对已经过镶嵌化的面片进行处理的顶点着色器”。特别是，我们可以在此将经镶嵌化处理的面片顶点投射到齐次裁剪空间。

对于四边形面片来讲，域着色器以曲面细分因子（还有一些来自常量外壳着色器所输出的每个面片的附加信息）、控制点外壳着色器所输出的所有面片控制点以及镶嵌化处理后的顶点位置参数坐标(u, v)作为输入。注意，域着色器给出的并不是镶嵌化处理后的实际顶点位置，而是这些点位于面片域空间（patch domain space）内的参数坐标(u, v)（见图14.4）。是否利用这些参数坐标以及控制点来求取真正的3D顶点位置，完全取决于用户自己。在以下代码中，我们将通过双线性插值（bilinear interpolation，其工作原理与纹理的线性过滤相似）来实现这一点。

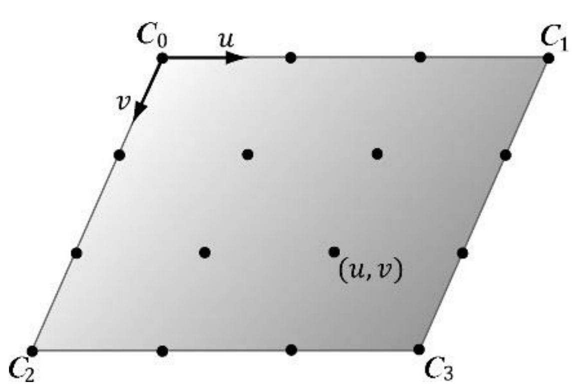


图14.4　对具有4个控制点的四边形面片进行细分，以生成坐标位于范围$[0,\ 1]^2$内的规范化uv空间中的16个顶点

```
struct DomainOut
{
  float4 PosH : SV_POSITION;
};

// 每当镶嵌器（tessellator）创建顶点时都会调用域着色器。
// 可以把它看作镶嵌处理阶段后的"顶点着色器"
[domain("quad")]
DomainOut DS(PatchTess patchTess,
       float2 uv : SV_DomainLocation,
       const OutputPatch<HullOut, 4> quad)
{
  DomainOut dout;

  // 双线性插值
  float3 v1 = lerp(quad[0].PosL, quad[1].PosL, uv.x);
  float3 v2 = lerp(quad[2].PosL, quad[3].PosL, uv.x);
  float3 p  = lerp(v1, v2, uv.y);

  float4 posW = mul(float4(p, 1.0f), gWorld);
  dout.PosH = mul(posW, gViewProj);

  return dout;
}
```

注意 Note

如图14.4所示，四边形面片的控制点顺序是一行紧接着一行（逐行）排列的。

域着色器处理三角形面片的方法与处理四边形面片的方法很相似，

只是把顶点用类型为`float3`的重心坐标(u, v, w)来表示，以此作为域着色器的输入（与重心坐标相关的数学内容请参见附录C.3节），从而取代参数坐标(u, v)。将三角形面片以重心坐标作为输入的原因，很可能是因为贝塞尔三角形面片都是用重心坐标来定义所导致的。

14.5　对四边形进行镶嵌化处理

作为本章中的第一个演示程序，我们在其中向渲染流水线提交了一个四边形面片，根据此面片与摄像机之间的距离对它进行镶嵌处理，再通过类似于之前示例中用于构造“山丘”的数学函数对生成的顶点进行平移。

按照如下方法创建存有4个控制点的顶点缓冲区。

```
void BasicTessellationApp::BuildQuadPatchGeometry()
{
  std::array<XMFLOAT3,4> vertices =
  {
    XMFLOAT3(-10.0f, 0.0f, +10.0f),
    XMFLOAT3(+10.0f, 0.0f, +10.0f),
    XMFLOAT3(-10.0f, 0.0f, -10.0f),
    XMFLOAT3(+10.0f, 0.0f, -10.0f)
  };

  std::array<std::int16_t, 4> indices = { 0, 1, 2, 3 };

  const UINT vbByteSize = (UINT)vertices.size() * sizeof(Vertex);
  const UINT ibByteSize = (UINT)indices.size() * sizeof(std::uint16_t);

  auto geo = std::make_unique<MeshGeometry>();
  geo->Name = "quadpatchGeo";

  ThrowIfFailed(D3DCreateBlob(vbByteSize, &geo->VertexBufferCPU));
  CopyMemory(geo->VertexBufferCPU->GetBufferPointer(), vertices.data(),
    vbByteSize);

  ThrowIfFailed(D3DCreateBlob(ibByteSize, &geo->IndexBufferCPU));
  CopyMemory(geo->IndexBufferCPU->GetBufferPointer(), indices.data(),
    ibByteSize);

  geo->VertexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
    mCommandList.Get(), vertices.data(), vbByteSize,
    geo->VertexBufferUploader);
```

```
  geo->IndexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
    mCommandList.Get(), indices.data(), ibByteSize,
    geo->IndexBufferUploader);

  geo->VertexByteStride = sizeof(XMFLOAT3);
  geo->VertexBufferByteSize = vbByteSize;
  geo->IndexFormat = DXGI_FORMAT_R16_UINT;
  geo->IndexBufferByteSize = ibByteSize;

  SubmeshGeometry quadSubmesh;
  quadSubmesh.IndexCount = 4;
  quadSubmesh.StartIndexLocation = 0;
  quadSubmesh.BaseVertexLocation = 0;

  geo->DrawArgs["quadpatch"] = quadSubmesh;

  mGeometries[geo->Name] = std::move(geo);
}
```

四边形面片的渲染项创建如下：

```
void BasicTessellationApp::BuildRenderItems()
{
  auto quadPatchRitem = std::make_unique<RenderItem>();
  quadPatchRitem->World = MathHelper::Identity4x4();
  quadPatchRitem->TexTransform = MathHelper::Identity4x4();
  quadPatchRitem->ObjCBIndex = 0;
  quadPatchRitem->Mat = mMaterials["whiteMat"].get();
  quadPatchRitem->Geo = mGeometries["quadpatchGeo"].get();
  quadPatchRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_4_CONTROL_
    POINT_PATCHLIST;
  quadPatchRitem->IndexCount = quadPatchRitem->Geo-
    >DrawArgs["quadpatch"].IndexCount;
  quadPatchRitem->StartIndexLocation =
    quadPatchRitem->Geo->DrawArgs["quadpatch"].StartIndexLocation;
  quadPatchRitem->BaseVertexLocation =
    quadPatchRitem->Geo->DrawArgs["quadpatch"].BaseVertexLocation;
  mRitemLayer[(int)RenderLayer::Opaque].push_back(quadPatchRitem.get());

  mAllRitems.push_back(std::move(quadPatchRitem));
}
```

现在我们把注意力转移到外壳着色器上。该外壳着色器与14.2.1节

及14.2.2节中介绍的比较相近，不同之处仅在于现在要根据观察点与网格的距离来确定曲面细分因子。这背后所隐藏的想法即网格与观察点之间距离较远时采用低模网格，并随着两者逐步接近而增加镶嵌的次数（继而增加了网格中的三角形个数，整个过程如图14.5所示）。

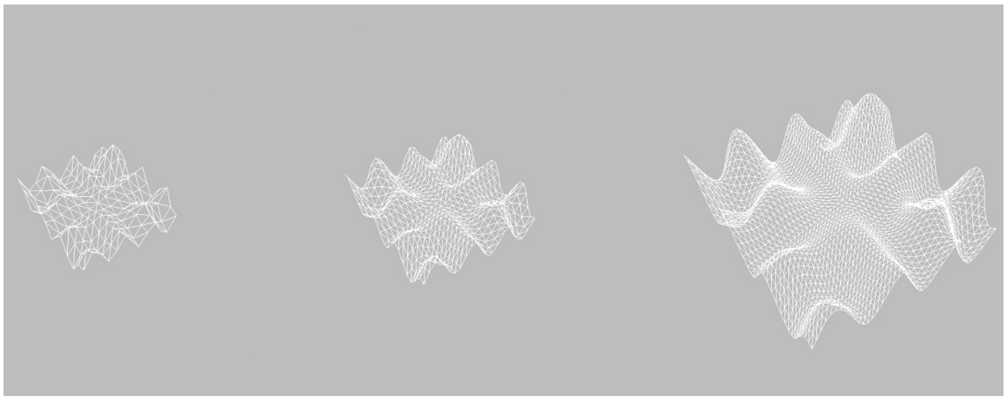


图14.5　网格细分的次数随着观察者与之距离的拉近而增加

```
struct VertexIn
{
  float3 PosL  : POSITION;
};

struct VertexOut
{
  float3 PosL  : POSITION;
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout;
  vout.PosL = vin.PosL;
  return vout;
}

struct PatchTess
{
  float EdgeTess[4]   : SV_TessFactor;
  float InsideTess[2] : SV_InsideTessFactor;
};
```

```
PatchTess ConstantHS(InputPatch<VertexOut, 4> patch, uint patchID :
    SV_PrimitiveID)
{
  PatchTess pt;

  float3 centerL = 0.25f*(patch[0].PosL +
            patch[1].PosL +
            patch[2].PosL +
            patch[3].PosL);

  float3 centerW = mul(float4(centerL, 1.0f), gWorld).xyz;

  float d = distance(centerW, gEyePosW);

  // 根据网格与观察点的距离来对面片进行镶嵌处理，如果d >= d1，则镶嵌份数为0；若d
 <= d0，那么镶
  // 嵌份数为64。[d0, d1]区间则定义了执行镶嵌操作的距离范围

  const float d0 = 20.0f;
  const float d1 = 100.0f;
  float tess = 64.0f*saturate( (d1-d)/(d1-d0) );

  // 对面片的各方面（边缘、内部）进行统一的镶嵌化处理

  pt.EdgeTess[0] = tess;
  pt.EdgeTess[1] = tess;
  pt.EdgeTess[2] = tess;
  pt.EdgeTess[3] = tess;

  pt.InsideTess[0] = tess;
  pt.InsideTess[1] = tess;

  return pt;
}

struct HullOut
{
  float3 PosL : POSITION;
};

[domain("quad")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(4)]
[patchconstantfunc("ConstantHS")]
[maxtessfactor(64.0f)]
HullOut HS(InputPatch<VertexOut, 4> p,
```

```
      uint i : SV_OutputControlPointID,
      uint patchId : SV_PrimitiveID)
{
  HullOut hout;
  hout.PosL = p[i].PosL;
  return hout;
}
```

仅是简单地镶嵌化处理还不足以为网格增添丰富的细节，因为新增的三角形仅仅是列于经过细分的面片（平面）之上。因此，我们一定要以某种方式来移动这些新增的顶点，使目标物体的形状更接近于预定的模型。而这些操作都是在域着色器中执行的。在这个演示程序中，我们以7.7.3节中介绍过的“山丘”模拟函数在*y*轴方向上对诸顶点进行偏移。

```
struct DomainOut
{
  float4 PosH : SV_POSITION;
};

// 每当镶嵌器创建顶点时都要调用域着色器
// 可以将它看作镶嵌环节后的“顶点着色器”
[domain("quad")]
DomainOut DS(PatchTess patchTess,
       float2 uv : SV_DomainLocation,
       const OutputPatch<HullOut, 4> quad)
{
  DomainOut dout;

  // 双线性插值
  float3 v1 = lerp(quad[0].PosL, quad[1].PosL, uv.x);
  float3 v2 = lerp(quad[2].PosL, quad[3].PosL, uv.x);
  float3 p  = lerp(v1, v2, uv.y);

  // 位移贴图（displacement mapping）
  p.y = 0.3f*( p.z*sin(p.x) + p.x*cos(p.z) );

  float4 posW = mul(float4(p, 1.0f), gWorld);
  dout.PosH = mul(posW, gViewProj);

  return dout;
}
```

```
float4 PS(DomainOut pin) : SV_Target
{
  return float4(1.0f, 1.0f, 1.0f, 1.0f);
}
```

14.6　三次贝塞尔四边形面片

在本节中，我们将描述三次贝塞尔四边形面片（cubic Bézier quad patch），并展示如何以更多的控制点来构建曲面。在讲解曲面之前，我们先来一睹贝塞尔曲线（Bézier curve）的“风采”。

14.6.1　贝塞尔曲线

现考虑我们有3个非共线的控制点$\boldsymbol{p}_0$、$\boldsymbol{p}_1$与$\boldsymbol{p}_2$。按如下方法借助这3个控制点来定义一条贝塞尔曲线。为求得曲线上的一点$\boldsymbol{p}(t)$，首先要用t在点$\boldsymbol{p}_0$与点$\boldsymbol{p}_1$之间以及点$\boldsymbol{p}_1$与点$\boldsymbol{p}_2$之间共进行两次线性插值，以此来分别获取其间的两个中间点：

$$\boldsymbol{p}_0^1 = (1-t)\boldsymbol{p}_0 + t\boldsymbol{p}_1$$

$$\boldsymbol{p}_1^1 = (1-t)\boldsymbol{p}_1 + t\boldsymbol{p}_2$$

接着，利用t在中间点$\boldsymbol{p}_0^1$与$\boldsymbol{p}_1^1$之间再次进行线性插值来求出曲线上一点$\boldsymbol{p}(t)$：

$$\begin{aligned}\boldsymbol{p}(t) &= (1-t)\boldsymbol{p}_0^1 + t\boldsymbol{p}_1^1 \\ &= (1-t)\left((1-t)\boldsymbol{p}_0 + t\boldsymbol{p}_1\right) + t\left((1-t)\boldsymbol{p}_1 + t\boldsymbol{p}_2\right) \\ &= (1-t)^2\boldsymbol{p}_0 + 2(1-t)t\boldsymbol{p}_1 + t^2\boldsymbol{p}_2\end{aligned}$$

换句话说，二次（二阶）贝塞尔曲线的参数方程是通过连续插值而推导出来的：

$$\boldsymbol{p}(t) = (1-t)^2\boldsymbol{p}_0 + 2(1-t)t\boldsymbol{p}_1 + t^2\boldsymbol{p}_2$$

同理，4个控制点$\boldsymbol{p}_0$、$\boldsymbol{p}_1$、$\boldsymbol{p}_2$以及$\boldsymbol{p}_3$则定义了3次（三阶）贝塞尔曲线，曲线上的点$\boldsymbol{p}(t)$同样可采用重复插值法求得。图14.6就演示了3次贝塞尔曲线的连续插值过程。首先，在4个给定控制点所定义的每条线段上进行第一次线性插值，求出第一波生成的三个中间点：

$$\boldsymbol{p}_0^1 = (1-t)\boldsymbol{p}_0 + t\boldsymbol{p}_1$$

$$\boldsymbol{p}_1^1 = (1-t)\boldsymbol{p}_1 + t\boldsymbol{p}_2$$

$$\boldsymbol{p}_2^1 = (1-t)\boldsymbol{p}_2 + t\boldsymbol{p}_3$$

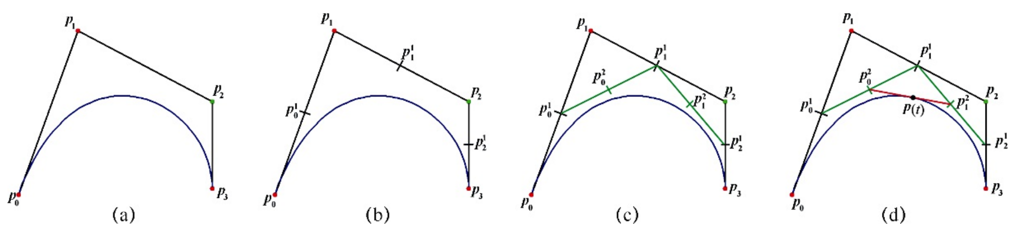


图14.6　为求出三次贝塞尔曲线上的点而进行重复插值运算（设插值因子为$t=0.5$）

（a）4个控制点以及它们所定义的三次贝塞尔曲线

（b）在控制点之间进行线性插值来计算第一批中间点

（c）在第一波中间点之间进行线性插值来生成第二批中间点

（d）在第二批中间点之间再次进行线性插值以求出三次贝塞尔曲线上的点

接下来，在第一批生成的中间点所连接的线段上进行线性插值，求取第二批生成的中间点：

$$\begin{aligned}\boldsymbol{p}_0^2 &= (1-t)\boldsymbol{p}_0^1 + t\boldsymbol{p}_1^1 \\ &= (1-t)^2\boldsymbol{p}_0 + 2(1-t)t\boldsymbol{p}_1 + t^2\boldsymbol{p}_2\end{aligned}$$

$$\begin{aligned}\boldsymbol{p}_1^2 &= (1-t)\boldsymbol{p}_1^1 + t\boldsymbol{p}_2^1 \\ &= (1-t)^2\boldsymbol{p}_1 + 2(1-t)t\boldsymbol{p}_2 + t^2\boldsymbol{p}_3\end{aligned}$$

最后，对第二批中间点进行线性插值以求出三次贝塞尔曲线上的点$\boldsymbol{p}(t)$：

$$\begin{aligned}\boldsymbol{p}(t) &= (1-t)\boldsymbol{p}_0^2 + t\boldsymbol{p}_1^2 \\ &= (1-t)\left((1-t)^2\boldsymbol{p}_0 + 2(1-t)t\boldsymbol{p}_1 + t^2\boldsymbol{p}_2\right) + t\left((1-t)^2\boldsymbol{p}_1 + 2(1-t)t\boldsymbol{p}_2 + t^2\boldsymbol{p}_3\right)\end{aligned}$$

将它化简为三次（三阶）贝塞尔曲线的参数方程：

$$\boldsymbol{p}(t) = (1-t)^3\boldsymbol{p}_0 + 3t(1-t)^2\boldsymbol{p}_1 + 3t^2(1-t)\boldsymbol{p}_2 + t^3\boldsymbol{p}_3 \qquad (14.1)$$

通常来讲，三次贝塞尔曲线就能够满足一般用户的需求了，因为它足够平滑，而且对曲线控制的自由度也比较高。当然，我们也可以继续以同样的递归方式进行重复插值，以获取更高阶的曲线。

事实证明，n阶贝塞尔曲线公式可以用伯恩斯坦基函数（Bernstein basis function）的形式来表示，其定义为：

$$B_i^n(t) = \frac{n!}{i!(n-i)!}t^i(1-t)^{n-i}$$

对于三阶曲线而言，它所对应的伯恩斯坦基函数分别为：

$$B_0^3(t) = \frac{3!}{0!(3-0)!}t^0(1-t)^{3-0} = (1-t)^3$$

$$B_1^3(t) = \frac{3!}{1!(3-1)!}t^1(1-t)^{3-1} = 3t(1-t)^2$$

$$B_2^3(t) = \frac{3!}{2!(3-2)!}t^2(1-t)^{3-2} = 3t^2(1-t)$$

$$B_3^3(t) = \frac{3!}{3!(3-3)!}t^3(1-t)^{3-3} = t^3$$

将这些结果与式（14.1）中的因子进行比对，我们就能把三次贝塞尔曲线方程写作：

$$\boldsymbol{p}(t)=\sum_{j=0}^{3}B_j^3(t)\boldsymbol{p}_j=B_0^3(t)\boldsymbol{p}_0+B_1^3(t)\boldsymbol{p}_1+B_2^3(t)\boldsymbol{p}_2+B_3^3(t)\boldsymbol{p}_3$$

我们可以运用导数的乘方与乘积运算法则来求出三次伯恩斯坦基函数的导数：

$$B_0^{3\prime}(t)=-3(1-t)^2$$

$$B_1^{3\prime}(t)=3(1-t)^2-6t(1-t)$$

$$B_2^{3\prime}(t)=6t(1-t)-3t^2$$

$$B_3^{3\prime}(t)=3t^2$$

因此，对3次贝塞尔曲线求导的结果为：

$$p'(t)=\sum_{j=0}^{3}B_j^{3\prime}(t)p_j=B_0^{3\prime}(t)p_0+B_1^{3\prime}(t)p_1+B_2^{3\prime}(t)p_2+B_3^{3\prime}(t)p_3$$

通过这些导数便可以很方便地计算出曲线上某点处的切向量。

注意 Note

网络上有演示贝塞尔曲线的相关小应用程序（applet）。我们可以对它进行设置或直接操纵其控制点，以此来观察这些因素与曲线形状的关联。

14.6.2　三次贝塞尔曲面

本节将自始至终围绕图14.7展开。考虑一个具有4×4控制点的面片。其每一行都含有4个控制点，因此就可以定义出4条三次贝塞尔曲线。第i行的贝塞尔曲线可表示为：

$$\boldsymbol{q}_i(u) = \sum_{j=0}^{3} B_j^3(u)\boldsymbol{p}_{i,j}$$

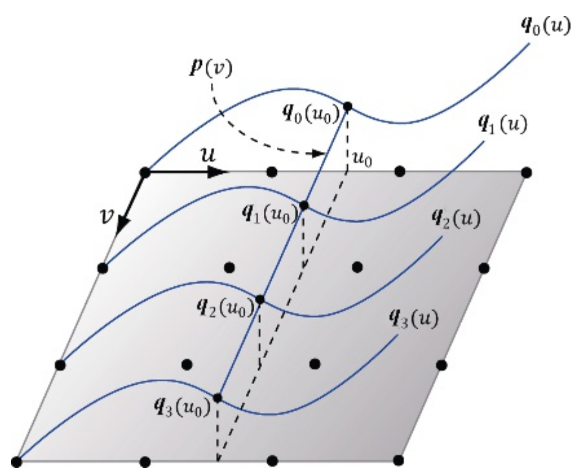


图14.7　构建一个贝塞尔曲面。此处进行了些合理的简化，使读者对该图示更容易理解——实际上，控制点通常并非都位于同一平面内，而所有$\boldsymbol{q}_i(u)$的位置也未必都如图中那样的理想（仅当曲线相同且每条曲线上的控制点位置都相同时才会出现图中的情况），而且，$\boldsymbol{p}(v)$往往不会是条直线，而是一条贝塞尔曲线

如果我们在每条贝塞尔曲线上的u_0处进行求值，便会获得4个点所构成的“列”，它必将经过每一条曲线。由此，我们就能通过这4个点来定义另外一条位于贝塞尔曲面（Bézier surface）上u_0处的贝塞尔曲线：

$$\boldsymbol{p}(v)=\sum_{i=0}^{3}B_i^3(v)\boldsymbol{q}_i(u_0)$$

如果现在将u设为一变量，便可使整条三次贝塞尔曲线移动而“扫”出一个三次贝塞尔曲面（cubic Bézier surface）：

$$\begin{aligned}\boldsymbol{p}(u,v)&=\sum_{i=0}^{3}B_i^3(v)\boldsymbol{q}_i(u)\\&=\sum_{i=0}^{3}B_i^3(v)\sum_{j=0}^{3}B_j^3(u)\boldsymbol{p}_{i,j}\end{aligned}$$

贝塞尔曲面的偏导数有助于计算出对应曲面处的切向量与法向量：

$$\frac{\partial \boldsymbol{p}}{\partial u}(u,v)=\sum_{i=0}^{3}B_i^3(v)\sum_{j=0}^{3}\frac{\partial B_j^3}{\partial u}(u)\boldsymbol{p}_{i,j}$$

$$\frac{\partial \boldsymbol{p}}{\partial v}(u,v)=\sum_{i=0}^{3}\frac{\partial B_i^3}{\partial v}(v)\sum_{j=0}^{3}B_j^3(u)\boldsymbol{p}_{i,j}$$

14.6.3　计算三次贝塞尔曲面的相关代码

在本节中，我们会给出用于计算三次贝塞尔曲面的相关代码。为了便于读者理解这些代码，我们将实现的求和表达式展开如下：

$$\boldsymbol{q}_0(u)=B_0^3(u)\boldsymbol{p}_{0,0}+B_1^3(u)\boldsymbol{p}_{0,1}+B_2^3(u)\boldsymbol{p}_{0,2}+B_3^3(u)\boldsymbol{p}_{0,3}$$

$$\boldsymbol{q}_1(u)=B_0^3(u)\boldsymbol{p}_{1,0}+B_1^3(u)\boldsymbol{p}_{1,1}+B_2^3(u)\boldsymbol{p}_{1,2}+B_3^3(u)\boldsymbol{p}_{1,3}$$

$$\boldsymbol{q}_2(u)=B_0^3(u)\boldsymbol{p}_{2,0}+B_1^3(u)\boldsymbol{p}_{2,1}+B_2^3(u)\boldsymbol{p}_{2,2}+B_3^3(u)\boldsymbol{p}_{2,3}$$

$$\boldsymbol{q}_3(u)=B_0^3(u)\boldsymbol{p}_{3,0}+B_1^3(u)\boldsymbol{p}_{3,1}+B_2^3(u)\boldsymbol{p}_{3,2}+B_3^3(u)\boldsymbol{p}_{3,3}$$

$$
\begin{aligned}
\boldsymbol{p}(u,v) &= B_0^3(v)\boldsymbol{q}_0(u) + B_1^3(v)\boldsymbol{q}_1(u) + B_2^3(v)\boldsymbol{q}_2(u) + B_3^3(v)\boldsymbol{q}_3(u) \\
&= B_0^3(v)\left[B_0^3(u)\boldsymbol{p}_{0,0} + B_1^3(u)\boldsymbol{p}_{0,1} + B_2^3(u)\boldsymbol{p}_{0,2} + B_3^3(u)\boldsymbol{p}_{0,3}\right] \\
&\quad + B_1^3(v)\left[B_0^3(u)\boldsymbol{p}_{1,0} + B_1^3(u)\boldsymbol{p}_{1,1} + B_2^3(u)\boldsymbol{p}_{1,2} + B_3^3(u)\boldsymbol{p}_{1,3}\right] \\
&\quad + B_2^3(v)\left[B_0^3(u)\boldsymbol{p}_{2,0} + B_1^3(u)\boldsymbol{p}_{2,1} + B_2^3(u)\boldsymbol{p}_{2,2} + B_3^3(u)\boldsymbol{p}_{2,3}\right] \\
&\quad + B_3^3(v)\left[B_0^3(u)\boldsymbol{p}_{3,0} + B_1^3(u)\boldsymbol{p}_{3,1} + B_2^3(u)\boldsymbol{p}_{3,2} + B_3^3(u)\boldsymbol{p}_{3,3}\right]
\end{aligned}
$$

以下代码就是由上述方程直接改写而得到的：

```
 float4 BernsteinBasis(float t)
{
  float invT = 1.0f - t;

  return float4( invT * invT * invT,  // B_0^3(t) = (1-t)^3
          3.0f * t * invT * invT,     // B_1^3(t) = 3t(1-t)^2
          3.0f * t * t * invT,        // B_2^3(t) = 3t^2(1-t)
          t * t * t );                // B_3^3(t) = t^3
}

float4 dBernsteinBasis(float t)
{
  float invT = 1.0f - t;

  return float4(
    -3 * invT * invT,               // B_0^3'(t) = -3(1-t)^2
    3 * invT * invT - 6 * t * invT, // B_1^3'(t) = 3(1-t)^2 - 6t(1-t)
    6 * t * invT - 3 * t * t,       // B_2^3'(t) = 6t(1-t) - 3t^2
    3 * t * t );                    // B_3^3'(t) = 3t^2
}

float3 CubicBezierSum(const OutputPatch bezpatch,
           float4 basisU, float4 basisV)
{
  float3 sum = float3(0.0f, 0.0f, 0.0f);
  sum = basisV.x * (basisU.x*bezpatch[0].PosL +
            basisU.y*bezpatch[1].PosL +
            basisU.z*bezpatch[2].PosL +
            basisU.w*bezpatch[3].PosL );

  sum += basisV.y * (basisU.x*bezpatch[4].PosL +
```

```
            basisU.y*bezpatch[5].PosL +
            basisU.z*bezpatch[6].PosL +
            basisU.w*bezpatch[7].PosL );

  sum += basisV.z * (basisU.x*bezpatch[8].PosL +
            basisU.y*bezpatch[9].PosL +
            basisU.z*bezpatch[10].PosL +
            basisU.w*bezpatch[11].PosL);

  sum += basisV.w * (basisU.x*bezpatch[12].PosL +
            basisU.y*bezpatch[13].PosL +
            basisU.z*bezpatch[14].PosL +
            basisU.w*bezpatch[15].PosL);

  return sum;
}
```

以上函数可用于求取$\boldsymbol{p}(u,v)$并计算其偏导数：

```
float4 basisU = BernsteinBasis(uv.x);
float4 basisV = BernsteinBasis(uv.y);

//   p(u,v)

float3 p = CubicBezierSum(bezPatch, basisU, basisV);

float4 dBasisU = dBernsteinBasis(uv.x);
float4 dBasisV = dBernsteinBasis(uv.y);

//   ∂p/∂u(u,v)

float3 dpdu = CubicBezierSum(bezPatch, dbasisU, basisV);

//   ∂p/∂v(u,v)

float3 dpdv = CubicBezierSum(bezPatch, basisU, dbasisV);
```

注意 Note

可以发现，我们把基函数（basis function）的计算结果传入了`CubicBezierSum`函数。由于$\boldsymbol{p}(u,v)$与其偏导数的求和形式相同，仅基函数不同，因此`CubicBezierSum`函数不仅能用来计算$\boldsymbol{p}(u,v)$，还可以用于求取其偏导数。

14.6.4　定义面片的几何形状

我们的顶点缓冲区存储着按下列方式来创建的16个控制点：

```
void BezierPatchApp::BuildQuadPatchGeometry()
{
  std::array<XMFLOAT3,16> vertices =
  {
    // 第0行
    XMFLOAT3(-10.0f, -10.0f, +15.0f),
    XMFLOAT3(-5.0f, 0.0f, +15.0f),
    XMFLOAT3(+5.0f, 0.0f, +15.0f),
    XMFLOAT3(+10.0f, 0.0f, +15.0f),

    // 第1行
    XMFLOAT3(-15.0f, 0.0f, +5.0f),
    XMFLOAT3(-5.0f, 0.0f, +5.0f),
    XMFLOAT3(+5.0f, 20.0f, +5.0f),
    XMFLOAT3(+15.0f, 0.0f, +5.0f),

    // 第2行
    XMFLOAT3(-15.0f, 0.0f, -5.0f),
    XMFLOAT3(-5.0f, 0.0f, -5.0f),
    XMFLOAT3(+5.0f, 0.0f, -5.0f),
    XMFLOAT3(+15.0f, 0.0f, -5.0f),

    // 第3行
    XMFLOAT3(-10.0f, 10.0f, -15.0f),
    XMFLOAT3(-5.0f, 0.0f, -15.0f),
    XMFLOAT3(+5.0f, 0.0f, -15.0f),
    XMFLOAT3(+25.0f, 10.0f, -15.0f)
```

```
  };

  std::array<std::int16_t, 16> indices =
  {
    0, 1, 2, 3,
    4, 5, 6, 7,
    8, 9, 10, 11,
    12, 13, 14, 15
  };

  const UINT vbByteSize = (UINT)vertices.size() * sizeof(Vertex);
  const UINT ibByteSize = (UINT)indices.size() * sizeof(std::uint16_t);

  auto geo = std::make_unique<MeshGeometry>();
  geo->Name = "quadpatchGeo";

  ThrowIfFailed(D3DCreateBlob(vbByteSize, &geo->VertexBufferCPU));
  CopyMemory(geo->VertexBufferCPU->GetBufferPointer(), vertices.data(),
    vbByteSize);

  ThrowIfFailed(D3DCreateBlob(ibByteSize, &geo->IndexBufferCPU));
  CopyMemory(geo->IndexBufferCPU->GetBufferPointer(), indices.data(),
    ibByteSize);

  geo->VertexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
    mCommandList.Get(), vertices.data(), vbByteSize, geo-
    >VertexBufferUploader);

  geo->IndexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
    mCommandList.Get(), indices.data(), ibByteSize, geo-
    >IndexBufferUploader);

  geo->VertexByteStride = sizeof(XMFLOAT3);
  geo->VertexBufferByteSize = vbByteSize;
  geo->IndexFormat = DXGI_FORMAT_R16_UINT;
  geo->IndexBufferByteSize = ibByteSize;

  SubmeshGeometry quadSubmesh;
  quadSubmesh.IndexCount = (UINT)indices.size();
  quadSubmesh.StartIndexLocation = 0;
  quadSubmesh.BaseVertexLocation = 0;

  geo->DrawArgs["quadpatch"] = quadSubmesh;

  mGeometries[geo->Name] = std::move(geo);
}
```

注意 Note

这里并没有严格地限定控制点一定要按等距排列为均匀的栅格。

按下列方式为四边形面片创建相应的渲染项：

```
void BezierPatchApp::BuildRenderItems()
{
  auto quadPatchRitem = std::make_unique<RenderItem>();
  quadPatchRitem->World = MathHelper::Identity4x4();
  quadPatchRitem->TexTransform = MathHelper::Identity4x4();
  quadPatchRitem->ObjCBIndex = 0;
  quadPatchRitem->Mat = mMaterials["whiteMat"].get();
  quadPatchRitem->Geo = mGeometries["quadpatchGeo"].get();
  quadPatchRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_16_CONTROL_
    POINT_PATCHLIST;
  quadPatchRitem->IndexCount = quadPatchRitem->Geo-
    >DrawArgs["quadpatch"].IndexCount;
  quadPatchRitem->StartIndexLocation =
    quadPatchRitem->Geo->DrawArgs["quadpatch"].StartIndexLocation;
  quadPatchRitem->BaseVertexLocation =
    quadPatchRitem->Geo->DrawArgs["quadpatch"].BaseVertexLocation;
  mRitemLayer[(int)RenderLayer::Opaque].push_back(quadPatchRitem.get());

  mAllRitems.push_back(std::move(quadPatchRitem));
}
```

图14.8所示即为贝赛尔曲面演示程序的效果。

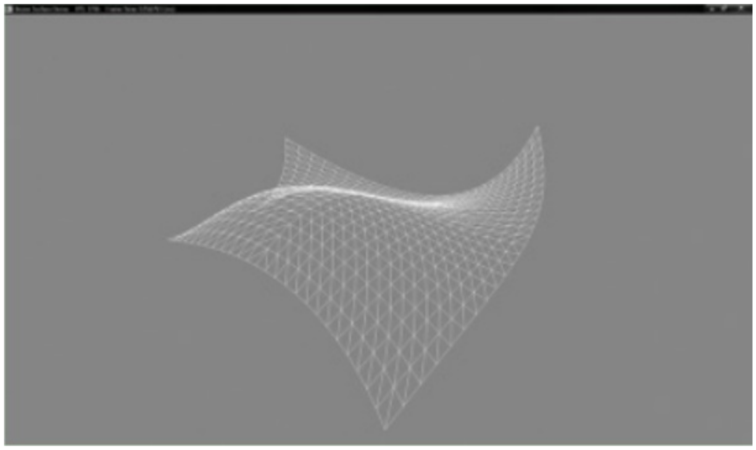


图14.8　贝塞尔曲面演示程序的效果

14.7 小结

1．曲面细分是渲染流水线中的一个可选阶段。它由外壳着色器、镶嵌器与域着色器构成。其中，外壳着色器与域着色器是可编程的，镶嵌器则全权交由硬件管理。

2．硬件曲面细分有助于节约内存资源。有了这项技术，我们就能仅存储低模网格，并按照需求动态地运用镶嵌化处理手段为网格增添细节。另外，像动画与物理模拟所需的计算工作，都可以在曲面细分之前的低模网格上以较低频次开展。最后要提到的是，连续LOD算法现在已经完全可以在GPU上实现，这在硬件曲面细分技术诞生之前是必须通过CPU来实现的。

3．在开启曲面细分功能，并向渲染流水线提交控制点后，前文中那几种新介绍的图元类型就该一展身手了。Direct3D 12支持具有1～32个控制点的面片，它们分别由枚举类型
D3D_PRIMITIVE_1_CONTROL_POINT_PATCH...
D3D_PRIMITIVE_32_CONTROL_POINT_PATCH来表示。

4．使用曲面细分技术时，顶点着色器就会以控制点作为输入，并针对每个控制点执行相应的动画或物理模拟计算。外壳着色器由常量外壳着色器与控制点外壳着色器组成。常量外壳着色器对每个面片都逐一进行计算，并输出面片所对应的曲面细分因子，该因子会指导镶嵌器对面片执行镶嵌化处理，将其划分为特定份数。对了，除了输出曲面细分

因子外，常量外壳着色器还能输出一些面片的其他可选数据。控制点外壳着色器以大量控制点作为输入与输出，每当有控制点输出时都会调用它一次。一般来讲，控制点外壳着色器会修改输入面片的曲面表示方式。例如，若向外壳着色器阶段输入具有3个控制点的三角形，它便可能输出拥有10个控制点的贝塞尔三角形面片。

5．曲面细分阶段会在创建每一个顶点时调用域着色器。尽管在开启曲面细分功能后，顶点着色器充当着“处理每一个控制点的顶点着色器”，但实质上外壳着色器才是“针对镶嵌化面片顶点进行处理的顶点着色器”。特别是，我们在此阶段便可以将镶嵌处理后的面片顶点投射到齐次裁剪空间，或针对这些顶点执行其他操作。

6．如果我们不对某物体进行镶嵌化处理（例如曲面细分因子非常接近于1的时候），就不要在渲染此物体时开启曲面细分阶段，这会白白产生一定的开销。避免对“占地”小于8像素的三角形执行镶嵌化处理。尽量将所有的镶嵌化物体一次性绘制出来，以避免在一帧中往复开关曲面细分功能。另外，在外壳着色器中可使用背面剔除与视锥体剔除这些优化手段，以便丢弃用户根本看不到的镶嵌面片。

7．通过参数方程来指定贝塞尔曲线与贝塞尔曲面，借此便可以描述出相应的平滑曲线与曲面。此二者的“形状”实则由控制点来加以操纵。除了能够直接绘制出平滑的曲面之外，贝塞尔曲面还可以用于一些流行的硬件曲面细分算法，如PN三角形方法与Catmull-Clark逼近型细分曲面方法（Catmull-Clark approximation）。

14.8　练习

1．重新编写“Basic Tessellation”（基本曲面细分）演示程序，但是这一次要对三角形面片而非四边形面片进行镶嵌处理。

2．根据正二十面体与观察点的距离关系将其镶嵌细分为一个球体。

3．修改“Basic Tessellation”演示程序，对平面四边形进行固定配置的曲面细分处理（fixed tessellation，如，不随观察点与物体间的距离修改细分参数）。尝试不同的边缘细分因子与内部细分因子，直到找到最为理想的设置参数。

4．探索非整型曲面细分。即尝试在“Basic Tessellation”演示程序中使用：

```
[partitioning("fractional_even")]
[partitioning("fractional_odd")]
```

5．为二次贝塞尔曲线计算伯恩斯坦基函数$B_0^2(t)$、$B_1^2(t)$与$B_2^2(t)$，再分别求出相应的导数$B_0^{2\prime}(t)$、$B_1^{2\prime}(t)$与$B_2^{2\prime}(t)$。最后，对二次贝塞尔曲面的参数方程求导。

6．尝试通过修改“Bézier Patch”（贝塞尔面片）演示程序中的控制点来改变其中的贝塞尔曲面。

7．用附有9个控制点的二次贝塞尔曲面重新实现“Bézier Patch”演示

程序。

8．修改“Bézier Patch”演示程序，利用光照使其中的贝塞尔曲面表现出明暗变化。为此，我们需要在域着色器中计算顶点法线。而位于顶点处的法线可由此曲面点处坐标的偏导数叉积求得。

9．研究并实现贝塞尔三角形面片。

---

[1] 曲面细分或称细分曲面是一种可将粗糙几何体网格细化的技术，英文原为subdivision surface。镶嵌（tessellation）则是可实现此技术的具体手段（如文中所述亦可实现LOD等），因此此处将“镶嵌”称为“曲面细分”实为化用。这里为便于区分整个“曲面细分阶段”与其中的“镶嵌器阶段”实现细节分别采用两种译法，请读者注意。

[2] `D3D_PRIMITIVE_TOPOLOGY`枚举项描述的是输入装配阶段中的顶点类型，而`D3D_PRIMITIVE`枚举项则描述的是外壳着色器的输入图元类型。

[3] Direct3D 12亦是如此，参见《Constants》（dn903792）中的`D3D12_TESSELLATOR_MAX_TESSELLATION_FACTOR`常量定义。

[4] silhouette edge的定义通常是外向平面法线垂直于观察向量的点在投影平面内的集合，也就是正面朝向平面与背面朝向平面间的交界线，即可见物体的轮廓线。结合上下文里的“朝向（orientation）”一词，译者认为作者这里所说的意思为正对观察者的三角形细节要丰富于侧面非正

对观察者的那些三角形。

[5] 用户向目标物体移动过程中，随距离变换会触发不同LOD的曲面细分，若取文中的整数因子，则细分之间物体细节变化过大，容易看出物体的变化过程。比如明明是一块棱角分明的石头，走了两步也许就变成球体了……除了文中介绍的3种`partitioning`参数之外，还有一种`pow2`因子，即因子值为2的次方。

[6] `outputtopology`参数还有一个point选项。

[7] 此阶段（tessellation stage，有时也写作tessellator stage）起主要作用的是镶嵌器。由于现在是将曲面细分阶段（官方名亦为tessellation stage，可见，二者的名称容易冲突）按顺序划分为外壳着色器（由常量外壳着色器与控制点外壳着色器组成）、镶嵌器、域着色器三个环节，所以此节名为“镶嵌器阶段”似乎更妥。

[8] 怀疑这是笔误。因为外壳着色器没有执行镶嵌处理，这一工作是由镶嵌器实现，而且还着重说明“已经过镶嵌化的面片（tessellated patch）”，结合域着色器在镶嵌器之后这一位置关系，再阅读代码中实际进行坐标变换的着色器，怀疑这里所述的应当为域着色器。

第三部分　主题篇

在本部分中，我们的重点是通过Direct3D实现几组3D应用来演示多种相关技术，例如渲染天空场景、环境光遮蔽（ambient occlusion，也有译作环境光吸收等）、角色动画（character animation）、拾取（picking）、环境贴图（environment mapping，也有译作环境映射）、法线贴图（normal mapping，也有译作法线映射）以及阴影贴图（shadow mapping，也有译作阴影映射）。

第15章“构建第一人称视角的摄像机与动态索引”本章将展示怎样按第一人称的游戏视角来设计一个摄像机系统。示例中，我们会通过键盘与鼠标的输入操作来控制此摄像机。除此之外，本章还会介绍Direct3D 12中新引进的动态索引（dynamic indexing）技术，以此对着色器中的纹理对象数组进行动态索引。

第16章“实例化与视锥体剔除”实例化（instancing）是一种硬件支持技术，可以针对同一几何图形以不同参数（意即在场景中的不同位置或具有不同的颜色）的绘制情况进行优化。视锥体剔除也是一种优化技术，当整个物体都处于虚拟摄像机的视野之外时，我们就可利用此技术将已提交至渲染管线的对应物体完全丢弃。我们还会在此章展示如何计算网格的包围盒与包围球。

第17章“拾取”本章会讲述如何来确定用户通过鼠标选取的特定3D

对象（或3D图元）。3D游戏与3D应用中经常会用到此技术，使用户利用鼠标与3D环境交互。

第18章“立方体贴图”本章将演示如何将环境贴图映射到任意形状的网格之上，还会通过环境图来为天空球（sky-sphere）贴图。

第19章“法线贴图”本章将展示如何通过使用法线图（normal map，即存有法向量的纹理）来为实时光照效果增添更加丰富的细节。由于法线图中的表面法线比逐顶点法线（per-vertex normal）的粒度更细，因此光照效果也更加真实。

第20章“阴影贴图”阴影贴图是一种实时的阴影绘制技术，借此可以将阴影渲染到任意形状的几何体之上（即不仅限于平面阴影）。本章还会介绍投影纹理的工作原理。

第21章“环境光遮蔽”在使场景变得更加真实的过程中，光照扮演着极为重要的角色。本章会根据场景中某点对入射光被周围物体遮挡的程度来改良光照方程中的环境光项。

第22章“四元数”本章将介绍称为四元数（quaternion）的数学对象（mathematical object），将展示用单位四元数表示旋转，并用简便的方法对它进行插值，以此来简洁地实现插值以及旋转操作。一旦可以实现上述操作，我们便可创建3D动画。

第23章“角色动画”本章涵盖了角色动画理论，并展示怎样通过一个复杂的行走动画来驱动人形游戏角色。

第15章　构建第一人称视角的摄像机与动态索引

在本章中，我们将分别探讨两个略显简短的主题。首先，我们要设计一个如同第一人称游戏中那样的摄像机系统，并用它来替代之前演示程序中的旋转摄像机系统（orbiting camera system）。其次，我们会介绍Direct3D 12中一种名为动态索引（dynamic indexing，自着色器模型5.1开始才有的新功能）的新技术，由此便可以对纹理对象数组（如`Texture2D gDiffuseMap[n]`）进行动态索引了。这与我们在第12章中索引的特殊纹理数组对象（`Texture2DArray`）有些相似，但与之不同的是，动态索引可以处理由不同大小与类型的纹理所构成的数组，因此比`Texture2DArray`资源灵活得多。

学习目标：

1．复习观察空间变换（view space transformation，也作取景变换）的相关数学知识。

2．能够指出第一人称摄像机具有代表性的功能。

3．学习如何来实现第一人称摄像机。

4．理解怎样对纹理数组进行动态索引。

15.1　重温取景变换

如图15.1所示，观察空间（view space，也有译作观察坐标系、视图空间等）是附属于摄像机的坐标系。此空间中，摄像机位于坐标系原点，沿z轴正方向观察，x轴指向摄像机右侧，y轴指向摄像机的正上方。我们能够用它来取代场景中的顶点相对于世界空间（world space，也有译作世界坐标系）所描述的位置关系，以便在渲染流水线的后续阶段中以摄像机坐标系来描述这些顶点。从世界空间至观察空间的坐标变换称为取景变换（view transform，也作观察变换等），对应的矩阵则称作观察矩阵（view matrix）。

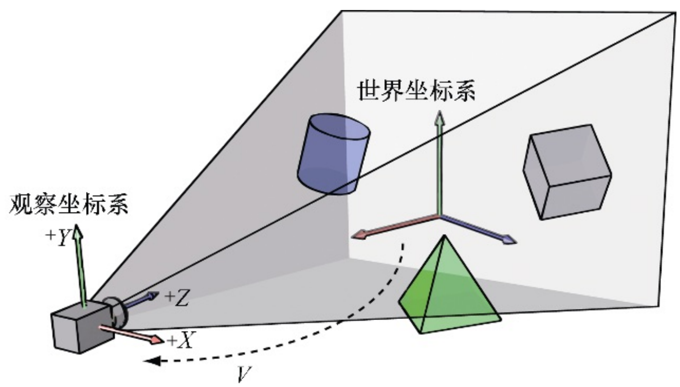


图15.1　摄像机坐标系。摄像机在自己专属的坐标系中，位于坐标系原点并沿z轴正方向观察

如果$\boldsymbol{Q}_W=(Q_x,\ Q_y,\ Q_z,\ 1)$、$\boldsymbol{u}_W=(u_x,\ u_y,\ u_z,\ 0)$、$\boldsymbol{v}_W=(v_x,\ v_y,\ v_z,\ 0)$且$\boldsymbol{w}_W=(w_x,\ w_y,\ w_z,\ 0)$分别表示观察空间中的原点、$x$轴、$y$轴与$z$轴相对于世界空间的齐次坐标，则根据3.4.3节可知，从观察空间至世界空间的坐标变换矩阵为：

$$\boldsymbol{W}=\begin{bmatrix}u_x & u_y & u_z & 0\\ v_x & v_y & v_z & 0\\ w_x & w_y & w_z & 0\\ Q_x & Q_y & Q_z & 1\end{bmatrix}$$

然而，这却并不是我们所期待的变换。刚好相反，我们需要的是由世界空间到观察空间的变换。再次回顾3.4.5节可知，这种逆变换通过上述矩阵的逆即可实现，因此可利用$\boldsymbol{W}^{-1}$将坐标从世界空间变换到观察空间。

一般来讲，世界坐标系与观察坐标系的差别仅在于位置与朝向，因此可直观地记作$\boldsymbol{W}=\boldsymbol{R}\boldsymbol{T}$（即将世界矩阵分解为一个旋转矩阵与一个平移矩阵的乘积）。这使得逆变换的计算更为简便：

$$\boldsymbol{V}=\boldsymbol{W}^{-1}=(\boldsymbol{R}\boldsymbol{T})^{-1}=\boldsymbol{T}^{-1}\boldsymbol{R}^{-1}=\boldsymbol{T}^{-1}\boldsymbol{R}^{\mathrm{T}}$$

$$=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ -Q_x & -Q_y & -Q_z & 1\end{bmatrix}\begin{bmatrix}u_x & v_x & w_x & 0\\ u_y & v_y & w_y & 0\\ u_z & v_z & w_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}=\begin{bmatrix}u_x & v_x & w_x & 0\\ u_y & v_y & w_y & 0\\ u_z & v_z & w_z & 0\\ -\boldsymbol{Q}\cdot\boldsymbol{u} & -\boldsymbol{Q}\cdot\boldsymbol{v} & -\boldsymbol{Q}\cdot\boldsymbol{w} & 1\end{bmatrix}$$

因此，观察矩阵形如：

$$\boldsymbol{V}=\begin{bmatrix}u_x & v_x & w_x & 0\\ u_y & v_y & w_y & 0\\ u_z & v_z & w_z & 0\\ -\boldsymbol{Q}\cdot\boldsymbol{u} & -\boldsymbol{Q}\cdot\boldsymbol{v} & -\boldsymbol{Q}\cdot\boldsymbol{w} & 1\end{bmatrix} \qquad (15.1)$$

同所有的坐标变换一样，我们并没有移动场景中的任何物体。由于我们用摄像机空间的参考标架（frame of reference）取代了世界空间的参考标架，以致场景中物体的坐标才纷纷发生了改变。

15.2　摄像机类

为了封装摄像机部分的相关代码，我们就要定义并实现Camera类。该摄像机类中的数据存储了两种关键信息。第一种为此类中定义的$position$、$right$、up与$look$向量，分别是以世界空间坐标表示的观察空间坐标系的原点、x轴、y轴、z轴。第二种为视锥体的属性。我们可以把摄像机的镜头（lens of camera）当作视锥体的定义（即视锥体的视场角以及近平面、远平面）。该类中实现的大都是功能琐碎（trivial，如简单的存取方法）的方法，其中数据成员和方法的概述可参见以下代码中的注释。在15.3节中，我们会对摄像机类中的关键方法进行讲解。

```
class Camera
{
public:

    Camera();
    ~Camera();

    // 获取及设置世界(空间中)摄像机的位置
    DirectX::XMVECTOR GetPosition()const;
    DirectX::XMFLOAT3 GetPosition3f()const;
    void SetPosition(float x, float y, float z);
    void SetPosition(const DirectX::XMFLOAT3& v);

    // 获取摄像机的基向量
    DirectX::XMVECTOR GetRight()const;
    DirectX::XMFLOAT3 GetRight3f()const;
    DirectX::XMVECTOR GetUp()const;
    DirectX::XMFLOAT3 GetUp3f()const;
    DirectX::XMVECTOR GetLook()const;
    DirectX::XMFLOAT3 GetLook3f()const;

    // 获取视锥体的属性
    float GetNearZ()const;
    float GetFarZ()const;
    float GetAspect()const;
```

```
    float GetFovY()const;
    float GetFovX()const;

    // 获取用观察空间坐标表示的近、远平面的大小
    float GetNearWindowWidth()const;
    float GetNearWindowHeight()const;
    float GetFarWindowWidth()const;
    float GetFarWindowHeight()const;

    // 设置视锥体
    void SetLens(float fovY, float aspect, float zn, float zf);

    // 通过LookAt方法的参数来定义摄像机空间
    void LookAt(DirectX::FXMVECTOR pos,
          DirectX::FXMVECTOR target,
          DirectX::FXMVECTOR worldUp);
    void LookAt(const DirectX::XMFLOAT3& pos,
          const DirectX::XMFLOAT3& target,
          const DirectX::XMFLOAT3& up);

    // 获取观察矩阵与投影矩阵
    DirectX::XMMATRIX GetView()const;
    DirectX::XMMATRIX GetProj()const;

    DirectX::XMFLOAT4X4 GetView4x4f()const;
    DirectX::XMFLOAT4X4 GetProj4x4f()const;

    // 将摄像机按距离d进行左右平移（Strafe）或前后移动（Walk）
    void Strafe(float d);
    void Walk(float d);

    // 将摄像机进行旋转
    void Pitch(float angle);
    void RotateY(float angle);

    // 修改摄像机的位置与朝向之后，调用此函数来重新构建观察矩阵
    void UpdateViewMatrix();

private:

    // 摄像机坐标系相对于世界空间的坐标
    DirectX::XMFLOAT3 mPosition = { 0.0f, 0.0f, 0.0f };
    DirectX::XMFLOAT3 mRight = { 1.0f, 0.0f, 0.0f };
    DirectX::XMFLOAT3 mUp = { 0.0f, 1.0f, 0.0f };
    DirectX::XMFLOAT3 mLook = { 0.0f, 0.0f, 1.0f };

    // 缓存视锥体的属性
```

```
    float mNearZ = 0.0f;
    float mFarZ = 0.0f;
    float mAspect = 0.0f;
    float mFovY = 0.0f;
    float mNearWindowHeight = 0.0f;
    float mFarWindowHeight = 0.0f;

    bool mViewDirty = true;

    // 缓存观察矩阵与投影矩阵
    DirectX::XMFLOAT4X4 mView = MathHelper::Identity4x4();
    DirectX::XMFLOAT4X4 mProj = MathHelper::Identity4x4();
};
```

注意 Note

Camera.h/Camera.cpp文件都位于Common文件夹中。

15.3 摄像机类中的方法实现选讲

本节内容不涉及摄像机类中的get/set简单存取方法，而是仅讨论其中比较重要的方法。

15.3.1 返回XMVECTOR类型变量的方法

首先要谈及的是摄像机类中各“get”方法返回XMVECTOR类型变量的问题。这样做其实只是为了使用方便而已，如此一来，若用户需要获取XMVECTOR类型的数据之时，则无需在代码自行转换：

```
XMVECTOR Camera::GetPosition()const
{
  return XMLoadFloat3(&mPosition);
}

XMFLOAT3 Camera::GetPosition3f()const
{
  return mPosition;
}
```

15.3.2 SetLens方法

我们可以认为摄像机的镜头即视锥体，因为它控制着观察者的视野。所以，我们在缓存视锥体属性以及构建投影矩阵时就要用到的SetLens方法：

```
void Camera::SetLens(float fovY, float aspect, float zn, float zf)
{
```

```
  // 缓存视锥体属性
  mFovY = fovY;
  mAspect = aspect;
  mNearZ = zn;
  mFarZ = zf;

  mNearWindowHeight = 2.0f * mNearZ * tanf( 0.5f*mFovY );
  mFarWindowHeight = 2.0f * mFarZ * tanf( 0.5f*mFovY );

  XMMATRIX P = XMMatrixPerspectiveFovLH(mFovY, mAspect, mNearZ, mFarZ);
  XMStoreFloat4x4(&mProj, P);
}
```

15.3.3 推导视锥体信息

正如刚刚所看到的头文件定义，我们不仅缓存了垂直视场角，还提供了额外的方法以推导视锥体的水平视场角。除此之外，我们还给出了特定方法返回近平面与远平面处视锥体的宽度与高度，在我们需要这些数据的时候，自然就会知道其妙处了。这些方法的实现完全依赖于三角学，如果对下列公式有疑问，读者可回顾5.6.3节。

```
float Camera::GetFovX()const
{
  float halfWidth = 0.5f*GetNearWindowWidth();
  return 2.0f*atan(halfWidth / mNearZ);
}

float Camera::GetNearWindowWidth()const
{
  return mAspect * mNearWindowHeight;
}

float Camera::GetNearWindowHeight()const
{
  return mNearWindowHeight;
}

float Camera::GetFarWindowWidth()const
{
```

```
  return mAspect * mFarWindowHeight;
}

float Camera::GetFarWindowHeight()const
{
  return mFarWindowHeight;
}
```

15.3.4　与摄像机相关的变换操作

对于第一人称摄像机来讲，若忽略其碰撞检测（collision detection）功能，我们还需要能够做到：

1．使摄像机沿着观察（look）向量前后移动。这可以通过令摄像机的位置沿其观察向量进行平移来实现。

2．令摄像机沿着它的右（right）向量左右平移（strafe）。这可以通过使摄像机的位置沿其右向量进行平移来实现。

3．使摄像机以右向量为轴，绕其旋转来进行俯仰观察。这可以通过使用`XMMatrixRotationAxis`函数，令摄像机的观察向量与上（up）向量绕其右向量进行旋转来实现。

4．令摄像机绕着世界空间的*y*轴（假设*y*轴对应于世界空间的“向上”方向）向量旋转来观察左右两侧。这可以通过使用`XMMatrixRotationY`函数，令所有的基向量（basis vector）绕世界空间的*y*轴进行旋转来实现。

```
void Camera::Walk(float d)
{
```

```
  // mPosition += d*mLook
  XMVECTOR s = XMVectorReplicate(d);
  XMVECTOR l = XMLoadFloat3(&mLook);
  XMVECTOR p = XMLoadFloat3(&mPosition);
  XMStoreFloat3(&mPosition, XMVectorMultiplyAdd(s, l, p));

void Camera::Strafe(float d)
{
  // mPosition += d*mRight
  XMVECTOR s = XMVectorReplicate(d);
  XMVECTOR r = XMLoadFloat3(&mRight);
  XMVECTOR p = XMLoadFloat3(&mPosition);
  XMStoreFloat3(&mPosition, XMVectorMultiplyAdd(s, r, p));
}

void Camera::Pitch(float angle)
{
  // 以右向量为轴旋转上向量与观察向量
  XMMATRIX R = XMMatrixRotationAxis(XMLoadFloat3(&mRight), angle);

  XMStoreFloat3(&mUp,   XMVector3TransformNormal(XMLoadFloat3(&mUp), R));
  XMStoreFloat3(&mLook, XMVector3TransformNormal(XMLoadFloat3(&mLook), R))
;
}

void Camera::RotateY(float angle)
{
  // 绕世界空间的y轴旋转所有的基向量
  XMMATRIX R = XMMatrixRotationY(angle);

  XMStoreFloat3(&mRight,  XMVector3TransformNormal(XMLoadFloat3(&mRight), 
R));
  XMStoreFloat3(&mUp, XMVector3TransformNormal(XMLoadFloat3(&mUp), R));
  XMStoreFloat3(&mLook, XMVector3TransformNormal(XMLoadFloat3(&mLook), R))
;
}
```

15.3.5　构建观察矩阵

UpdateViewMatrix方法首先将摄像机的右向量（right）、上向量（up）与观察向量（look）分别重新进行正交规范化

（reorthonormalize，也有译作规范正交化等）处理。以此确保它们彼此正交，且都为单位长度。这样做很有必要，因为一连串的旋转操作以及累积的数值误差会使它们变为非正交规范向量。若果真如此，那么这3个向量表示的将不再是直角坐标系，而是一个斜坐标系（skewed coordinate system，也有译作非对称坐标系），这并非我们的本意。该方法的后续部分就是将这3个摄像机向量代入式（15.1）中，从而计算出观察变换矩阵。

```
void Camera::UpdateViewMatrix()
{
  if(mViewDirty)
  {
    XMVECTOR R = XMLoadFloat3(&mRight);
    XMVECTOR U = XMLoadFloat3(&mUp);
    XMVECTOR L = XMLoadFloat3(&mLook);
    XMVECTOR P = XMLoadFloat3(&mPosition);

    // 使摄像机的坐标向量彼此正交且保持单位长度
    L = XMVector3Normalize(L);
    U = XMVector3Normalize(XMVector3Cross(L, R));

    // U，L已互为正交规范化向量，所以不需要对下列叉积再进行规范化处理
    R = XMVector3Cross(U, L);

    // 填写观察矩阵中的元素
    float x = -XMVectorGetX(XMVector3Dot(P, R));
    float y = -XMVectorGetX(XMVector3Dot(P, U));
    float z = -XMVectorGetX(XMVector3Dot(P, L));

    XMStoreFloat3(&mRight, R);
    XMStoreFloat3(&mUp, U);
    XMStoreFloat3(&mLook, L);

    mView(0, 0) = mRight.x;
    mView(1, 0) = mRight.y;
    mView(2, 0) = mRight.z;
    mView(3, 0) = x;

    mView(0, 1) = mUp.x;
    mView(1, 1) = mUp.y;
```

```
    mView(2, 1) = mUp.z;
    mView(3, 1) = y;

    mView(0, 2) = mLook.x;
    mView(1, 2) = mLook.y;
    mView(2, 2) = mLook.z;
    mView(3, 2) = z;

    mView(0, 3) = 0.0f;
    mView(1, 3) = 0.0f;
    mView(2, 3) = 0.0f;
    mView(3, 3) = 1.0f;

    mViewDirty = false;
  }
}
```

15.4　摄像机演示程序的若干注解

现在，我们就能从应用程序类（如本章即为CameraAndDynamicIndexingApp类）中移除如mPhi、mTheta、mRadius、mView与mProj这些与旋转摄像机系统相关的旧变量。另外，还需添加一个成员变量：

```
Camera mCamera;
```

当调整窗口大小时，我们就不必亲自重新构建透视投影矩阵了，只需将此任务委派给Camera类中的SetLens方法即可：

```
void CameraAndDynamicIndexingApp::OnResize()
{
  D3DApp::OnResize();

  mCamera.SetLens(0.25f*MathHelper::Pi, AspectRatio(), 1.0f, 1000.0f);
}
```

可以在OnKeyboardInput方法中处理键盘的输入，以此来移动摄像机：

```
void CameraAndDynamicIndexingApp::OnkeyboardInput(constGameTime&gt)
{
const float dt= gt.DeltaTime();

if(GetAsyncKeyState('W') & 0x8000)
  mCamera.Walk(10.0f*dt);

if(GetAsyncKeyState('S') & 0x8000)
  mCamera.Walk(-10.0f*dt);

if(GetAsyncKeyState('A') & 0x8000)
  mCamera.Strafe(-10.0f*dt);
```

```
if(GetAsyncKeyState('D') & 0x8000)
  mCamera.Strafe(10.0f*dt);
  mCamera.UpdateViewMatrix();
}
```

而在OnMouseMove方法中，我们以旋转摄像机的方式来调整其观察方向：

```
void CameraAndDynamicIndexingApp::OnMouseMove(WPARAM btnState, int x, int y)
{
  if( (btnState & MK_LBUTTON) != 0 )
  {
    // 根据鼠标的移动距离计算旋转角度，并使每个像素都按此角度的1/4进行旋转
    float dx = XMConvertToRadians(
      0.25f*static_cast(x - mLastMousePos.x));
    float dy = XMConvertToRadians(
      0.25f*static_cast(y - mLastMousePos.y));

    mCamera.Pitch(dy);
    mCamera.RotateY(dx);
  }

  mLastMousePos.x = x;
  mLastMousePos.y = y;
}
```

最后，出于渲染的目的，应当允许摄像机实例中的观察矩阵与投影矩阵可以被任意访问：

```
mCamera.UpdateViewMatrix();

XMMATRIX view = mCamera.GetView();
XMMATRIX proj = mCamera.GetProj();
```

图15.2所示为摄像机例程的示意图。

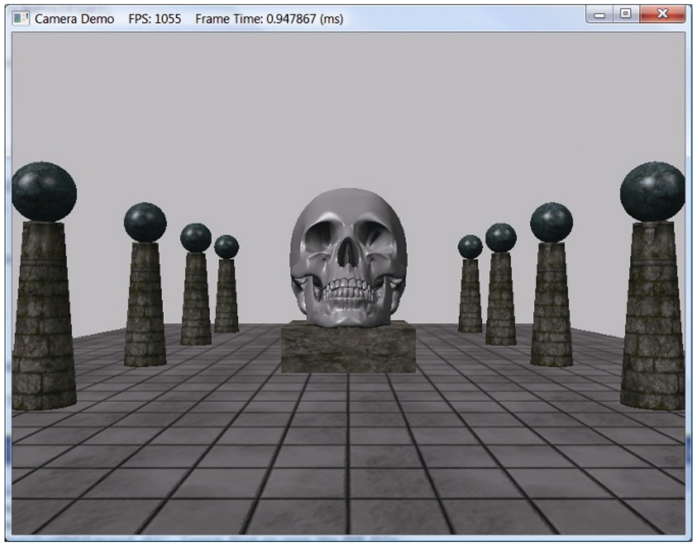


图15.2　摄像机例程的效果图。通过“W”“S”“A”“D”4个键来分别控制摄像机前、后、左、右4个方向的平移。按住鼠标左键并移动鼠标来“观察”不同的方向

15.5　动态索引

动态索引的概念比较简单，即在着色器程序中对资源数组进行动态地索引。在本章的演示程序中，所用的资源是纹理数组。指定索引的方法各式各样：

1．索引可以是常量缓冲区中的某个元素。

2．索引可以是如SV_PrimitiveID、SV_VertexID、SV_DispatchThreadID或SV_InstanceID等类似的系统ID。

3．索引可以通过计算求取。

4．索引可来自于纹理所存的数据。

5．索引也可以出自顶点结构体中的分量。

下列着色器语法声明了一个具有4个元素的纹理数组，并展示了怎样利用来自常量缓冲区中的索引来对该纹理数组进行索引。

```
cbuffer cbPerDrawIndex : register(b0)
{
  int gDiffuseTexIndex;
};

Texture2D gDiffuseMap[4] : register(t0);

float4 texValue = gDiffuseMap[gDiffuseTexIndex].Sample(
  gsamLinearWrap, pin.TexC);
```

对于这个演示程序来说，我们的目标是把每个渲染项所需配置的描述符数量降到最低。眼下，我们要为每个渲染项设置物体常量缓冲区、材质常量缓冲区以及漫反射纹理图的SRV（着色器资源视图）。要使描述符的数量降到最少，便需要令根签名的规模变得更小，这也就意味着每次绘制调用所需的开销也会随之减少。此时，若把动态索引与实例化（instancing，下一章的主题）两项技术搭配使用，则效果拔群。我们所用的策略如下。

1．创建一个存有所有材质数据的结构化缓冲区。即以结构化缓冲区代替常量缓冲区来存储其材质数据。我们可以在着色器程序中对结构化缓冲区进行索引。在绘制每一帧画面时，都将该结构化缓冲区与渲染流水线绑定一次，以令所有的材质都能被着色器程序所用。

2．通过为物体常量缓冲区添加`MaterialIndex`字段来指定本次绘制调用所用的材质索引。在着色器程序中，我们利用此字段（`gMaterialIndex`）来索引材质结构化缓冲区。

3．在绘制每一帧画面时，直接将场景中用到的全部纹理SRV描述符（以描述符表的形式）与渲染流水线一次性绑定，而不是像之前那样分别绑定每个渲染项的纹理SRV。

4．向材质数据结构体中添加`DiffuseMapIndex`字段，以指定与材质所关联的纹理图。据此，我们便可对上一步骤中与流水线相绑定的纹理数组进行索引。

经过这一系列配置，我们仅需为每个渲染项都设置一个物体常量缓

冲区。一旦实现了这些内容，我们就能通过`MaterialIndex`字段为绘制调用而获取相应的材质，并通过`DiffuseMapIndex`字段为绘制调用拾取所需的纹理。

注意 Note

前文提到，结构化缓冲区其实是一种由若干类型数据所构成的数组，可存于GPU端的显存之中，并通过着色器程序访问。由于我们仍需动态地更新材质，因此要使用上传缓冲区（upload buffer）而非默认缓冲区。而且在帧资源类中，将以材质结构化缓冲区取代之前的材质常量缓冲区，其创建过程如下：

```
struct MaterialData
{
  DirectX::XMFLOAT4 DiffuseAlbedo = { 1.0f, 1.0f, 1.0f, 1.0f };
  DirectX::XMFLOAT3 FresnelR0 = { 0.01f, 0.01f, 0.01f };
  float Roughness = 64.0f;

  // 用于纹理贴图
  DirectX::XMFLOAT4X4 MatTransform = MathHelper::Identity4x4();

  UINT DiffuseMapIndex = 0;
  UINT MaterialPad0;
  UINT MaterialPad1;
  UINT MaterialPad2;
};
MaterialBuffer = std::make_unique<UploadBuffer<MaterialData>>(
  device, materialCount, false);
```

除此之外，材质结构化缓冲区与材质常量缓冲区两者的代码差别不大。

接下来，根据着色器所需输入的新数据对根签名进行更新：

```
CD3DX12_DESCRIPTOR_RANGE texTable;
texTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 4, 0, 0);

// 根参数能够是描述符表、根描述符或根常量
CD3DX12_ROOT_PARAMETER slotRootParameter[4];

// 性能小提示：按变更频率由高至低排列
slotRootParameter[0].InitAsConstantBufferView(0);
slotRootParameter[1].InitAsConstantBufferView(1);
slotRootParameter[2].InitAsShaderResourceView(0, 1);
slotRootParameter[3].InitAsDescriptorTable(1, &texTable,
    D3D12_SHADER_VISIBILITY_PIXEL);

auto staticSamplers = GetStaticSamplers();

// 根签名由一系列根参数组成
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(4, slotRootParameter,
  (UINT)staticSamplers.size(), staticSamplers.data(),
  D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
```

到此为止，在绘制渲染项之前，我们就能在每帧中一次性绑定所有材质与纹理的SRV，而不必将每个渲染项依次进行绑定。而后，只需为每个渲染项设置其对应的物体常量缓冲区即可：

```
void CameraAndDynamicIndexingApp::Draw(const GameTimer& gt)
{
  ...
  auto passCB = mCurrFrameResource->PassCB->Resource();
  mCommandList->SetGraphicsRootConstantBufferView(1, passCB-
    >GetGPUVirtualAddress());

  // 绑定场景中要用到的所有材质。对于结构化缓冲区而言，我们能绕开描述符堆而直接
```

```
将其设置为根描述
  // 符
  auto matBuffer = mCurrFrameResource->MaterialBuffer->Resource();
  mCommandList->SetGraphicsRootShaderResourceView(2,
    matBuffer->GetGPUVirtualAddress());

  // 绑定场景中需要的所有纹理。可以发现，我们仅须指定表中的第一个描述符。而根签名将自行推断描述符表
  // 里到底含有多少个描述符
  mCommandList->SetGraphicsRootDescriptorTable(3,
    mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());

  DrawRenderItems(mCommandList.Get(), mOpaqueRitems);
  ...
}

void CameraAndDynamicIndexingApp::DrawRenderItems(
  ID3D12GraphicsCommandList* cmdList,
  const std::vector& ritems)
{
  ...
  // 针对每个渲染项...
  for(size_t i = 0; i < ritems.size(); ++i)
  {
    auto ri = ritems[i];
    ...

    cmdList->SetGraphicsRootConstantBufferView(0, objCBAddress);

    cmdList->DrawIndexedInstanced(ri->IndexCount, 1,
      ri->StartIndexLocation, ri->BaseVertexLocation, 0);
  }
}
```

可以看到，ObjectConstants结构体中已含有了MaterialIndex字段。为此字段设置的值应当与在材质常量缓冲区中引用对应材质数据所用的索引值相同：

```
// 更新物体常量缓冲区……
ObjectConstants objConstants;
XMStoreFloat4x4(&objConstants.World, XMMatrixTranspose(world));
XMStoreFloat4x4(&objConstants.TexTransform, XMMatrixTranspose(texTransform
));
objConstants.MaterialIndex = e->Mat->MatCBIndex;
```

按下列加粗字体的部分修改着色器代码以示范动态索引的用法：

```
// 此头文件包含光照所需的结构体与函数
#include "LightingUtil.hlsl"

// 每种材质所用到的不同常量数据
struct MaterialData
{
  float4   DiffuseAlbedo;
  float3   FresnelR0;
  float    Roughness;
  float4x4 MatTransform;
  uint     DiffuseMapIndex;
  uint     MatPad0;
  uint     MatPad1;
  uint     MatPad2;
};

// 一种只有着色器模型5.1+才支持的纹理数组。与Texture2DArray类型数组不同的是，此数组中所存纹
// 理的尺寸与格式可各不相同，这使它比一般的纹理数组更为灵活
Texture2D gDiffuseMap[4] : register(t0);

// 将此材质结构化缓冲区置于space1中，使纹理数组不会与这些资源相重叠。而纹理数组将占用寄存器t0，
// t1，…，t3中的space0空间
StructuredBuffer gMaterialData : register(t0, space1);

SamplerState gsamPointWrap        : register(s0);
SamplerState gsamPointClamp       : register(s1);
SamplerState gsamLinearWrap       : register(s2);
SamplerState gsamLinearClamp      : register(s3);
SamplerState gsamAnisotropicWrap  : register(s4);
SamplerState gsamAnisotropicClamp : register(s5);

// 每一帧都有所变化的常量数据
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorld;
  float4x4 gTexTransform;
  uint gMaterialIndex;
  uint gObjPad0;
  uint gObjPad1;
  uint gObjPad2;
};
```

```
// 绘制过程中所用的杂项常量数据
cbuffer cbPass : register(b1)
{
  float4x4 gView;
  float4x4 gInvView;
  float4x4 gProj;
  float4x4 gInvProj;
  float4x4 gViewProj;
  float4x4 gInvViewProj;
  float3 gEyePosW;
  float cbPerObjectPad1;
  float2 gRenderTargetSize;
  float2 gInvRenderTargetSize;
  float gNearZ;
  float gFarZ;
  float gTotalTime;
  float gDeltaTime;
  float4 gAmbientLight;

  // 对于每个以MaxLights 为光源数量最大值的对象来说，索引[0, NUM_DIR_LIGHTS)表示的是方向
  // 光源索引[NUM_DIR_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHTS)表示的是点光源
  // 索引 [NUM_DIR_LIGHTS+NUM_POINT_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHT+NUM_SPOT_
  // LIGHTS)则表示的是聚光灯光源
  Light gLights[MaxLights];
};

struct VertexIn
{
  float3 PosL    : POSITION;
  float3 NormalL : NORMAL;
  float2 TexC    : TEXCOORD;
};

struct VertexOut
{
  float4 PosH    : SV_POSITION;
  float3 PosW    : POSITION;
  float3 NormalW : NORMAL;
  float2 TexC    : TEXCOORD;
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout = (VertexOut)0.0f;
```

```
  // 获取材质数据
  MaterialData matData = gMaterialData[gMaterialIndex];

  // 把顶点变换到世界空间
  float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
  vout.PosW = posW.xyz;

  // 假设这里进行的是等比缩放，否则需要使用世界矩阵的逆转置矩阵进行计算
  vout.NormalW = mul(vin.NormalL, (float3x3)gWorld);

  // 将顶点变换到齐次裁剪空间
  vout.PosH = mul(posW, gViewProj);

  // 为三角形插值而输出顶点属性
  float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
  vout.TexC = mul(texC, matData.MatTransform).xy;

  return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
  // 获取材质数据
  MaterialData matData = gMaterialData[gMaterialIndex];
  float4 diffuseAlbedo = matData.DiffuseAlbedo;
  float3 fresnelR0 = matData.FresnelR0;
  float roughness = matData.Roughness;
  uint diffuseTexIndex = matData.DiffuseMapIndex;

  // 在数组中动态地查找纹理
  diffuseAlbedo *= gDiffuseMap[diffuseTexIndex].Sample(gsamLinearWrap, pin
.TexC);

  // 对法线插值可能造成其非规范化，因此要为它再次进行规范化处理
  pin.NormalW = normalize(pin.NormalW);

  // 经表面上一点反射向观察点的向量
  float3 toEyeW = normalize(gEyePosW - pin.PosW);

  // 光照项
  float4 ambient = gAmbientLight*diffuseAlbedo;

  Material mat = { diffuseAlbedo, fresnelR0, roughness };
  float4 directLight = ComputeLighting(gLights, mat, pin.PosW, pin.NormalW
,
    toEyeW);
```

```
  float4 litColor = ambient + directLight;

  // 从漫反射反照率获取alpha值的常用手段
  litColor.a = diffuseAlbedo.a;

  return litColor;
}
```

注意 Note

以上着色器代码显式地指明了寄存器空间：

```
StructuredBuffer gMaterialData : register(t0, space1);
```

若非如此，那么其默认的寄存器空间为space0。通过指定此空间，我们就可以使用着色器寄存器的其他维度，以防资源重叠。例如，我们可以凭下列方式将多种资源存于寄存器t0的不同空间之中：

```
Texture2D gDiffuseMap : register(t0, space0);
Texture2D gNormalMap : register(t0, space1);
Texture2D gShadowMap : register(t0, space2);
```

当数据为资源数组时，此法颇有成效。例如，下列具有四个元素的纹理数组占用了寄存器t0、t1、t2以及t3：

```
Texture2D gDiffuseMap[4] : register(t0);
```

由此，我们便可以推算出下一个可用的空闲寄存器为t4，或者直接使用一个新的寄存器空间而不必在此问题上思虑过多：

```
// 将结构化缓冲区存于space1中，这样一来，纹理数组就不会与之重叠了
```

```
// 纹理数组将会占用寄存器t0，t1，…， t3中的space0空间
  StructuredBuffer<MaterialData> gMaterialData : register(t0, space1);
```

在此，我们以动态索引的其他3种用法作为本节的结束。

1．将有着不同纹理的邻近网格合并为一个单独的渲染项，这样一来，仅需一次绘制调用就能把它们全部绘制出来。可以把这些网格的纹理与材质数据保存为顶点结构体中的一个属性。

2．在含有不同大小与不同格式纹理的单次渲染过程中，使用多纹理贴图技术（multitexturing）。

3．以系统值SV_InstanceID作为索引来实例化具有不同纹理与不同材质的渲染项。我们将在下一章中见到相关的示例。

15.6　小结

1．我们通过指定摄像机的位置与朝向来定义摄像机坐标系。该坐标系的位置是根据相对于世界坐标系的位置向量来加以确定的，而它的朝向则由其相对于世界坐标系的右向量、上向量和观察向量来指定。移动摄像机就相当于：相对于世界坐标系移动摄像机坐标系。

2．我们将与投影相关的数据纳入了摄像机类，因为透视投影矩阵可视为摄像机的“镜头”，以此便能调整视锥体的视场角、近平面与远平面。

3．沿着摄像机的观察向量前后平移摄像机即可轻松实现视角的前进与后退。按摄像机的右向量平移即可实现镜头的左右移动。至于俯仰观察，可以令摄像机绕其右向量旋转来完成。而向左、向右巡视可通过使所有的基向量绕世界空间的y轴旋转来实现。

4．动态索引是着色器模型5.1引入的新技术，它使我们可以对具有不同大小及格式纹理的数组进行动态索引。此技术的一项应用是：在我们绘制每一帧画面时，可一次性绑定所有的纹理描述符，随后，在像素着色器中对纹理数组进行动态索引来为像素找到它所对应的纹理。

15.7　练习

1．给出世界空间各坐标轴及其原点的世界坐标：$\boldsymbol{i}=(1,0,0)$、$\boldsymbol{j}=(0,1,0)$、$\boldsymbol{k}=(0,0,1)$和$\boldsymbol{O}=(0,0,0)$，以及观察空间诸坐标轴及其原点的世界坐标：$\boldsymbol{u}=(u_x,\ u_y,\ u_z)$、$\boldsymbol{v}=(v_x,\ v_y,\ v_z)$、$\boldsymbol{w}=(w_x,\ w_y,\ w_z)$和$\boldsymbol{Q}=(Q_x,\ Q_y,\ Q_z)$，利用点积推导出下列形式的观察矩阵：

$$\boldsymbol{V}=\begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ -\boldsymbol{Q}\cdot\boldsymbol{u} & -\boldsymbol{Q}\cdot\boldsymbol{v} & -\boldsymbol{Q}\cdot\boldsymbol{w} & 1 \end{bmatrix}$$

（还记得吗？要求出由世界空间到观察空间的变换矩阵，我们只需描述出世界空间各坐标轴及其原点相对于观察空间的坐标。接下来，再把这几个坐标作为观察矩阵的行向量。）

2．修改摄像机演示程序，使它支持“横滚”（roll）动作，就是使摄像机绕其观察向量旋转，令视角实现“侧滚翻“。这对于飞行类游戏而言是必不可少的因素。

3．假设现有这样一个场景：内含5个位置与纹理各不相同的正方体。创建一个网格存储这5个处于不同位置的立方体，再为它们统一创建一个渲染项。最后向顶点结构体添加一个字段用于索引纹理。举个例子，立方体0上的顶点应被映射为索引0所指向的纹理，因此立方体0将被纹理0所渲染。同理，立方体1上的顶点应被贴上索引1所指向的纹理，因此立方体1将被绘制为纹理1，并以此类推。在每一帧都把这5种

纹理与流水线绑定一次，并在像素着色器中用顶点结构体内的索引来选取对应的纹理。注意，我们要在一次绘制调用过程中，用五种不同的纹理来渲染这5个立方体。如果绘制调用（draw call）成为了应用程序中的瓶颈，不妨尝试像本练习所做的那样，将邻近的几何体合并到一个渲染项中，借此令程序的性能得到优化。

第16章　实例化与视锥体剔除

本章将介绍实例化与视锥体剔除的相关知识。实例化技术常用于对场景中同一对象反复绘制多次的情形。它所带来的优化效果显著，所以Direct3D专门为此提供了支持。视锥体剔除技术则是通过简单的测试将位于视锥体外的整组三角形从后续的处理流程中剔除出去。

学习目标：

1．学习如何实现硬件实例化。

2．熟悉包围体（bounding volume），了解这种辅助几何体备受青睐的原因以及它们的使用方法。

3．探索实现视锥体剔除技术。

16.1 硬件实例化

实例化技术常见于同一对象在场景中被绘制多次的情形，而每次绘制时，该物体的位置、朝向、缩放大小、材质乃至纹理可能都各不相同。下面是几个相关示例。

1．多次绘制几种稍有不同的树木模型来构筑森林。

2．多次渲染几种略有不同的小行星模型来搭建小行星带。

3．多次绘制几种稍有差异的人物模型来营造熙攘的人群。

要令每个实例都各自维护一套顶点数据与索引数据将极大地耗费系统资源。因此，我们以存储一份相对于其局部空间的几何体副本（即顶点列表与索引列表）的方法来加以取代。这样一来，在多次绘制同一对象的过程中，每次只要按具体需求使用不同的世界矩阵与材质即可。

尽管此策略可节省内存，但为绘制每个对象而调用的API开销仍然可观。即针对每一个对象，我们还必须为其设置独有的材质和世界矩阵，再执行绘制命令。尽管Direct3D 12经重新设计，已将Direct3D 11在绘制调用过程中所执行大部分API的开销降到最低，但少量负载依然存在。Direct3D实例化API使我们可以通过一次绘制调用构造出一个对象的多个实例。再者，有了动态索引（前一章的主题）的辅助，实例化技术将比Direct3D 11时期更具灵活性。

注意 Note

为什么总是对API的开销念念不忘呢？对于Direct3D 11来说，由于API的开销而致使应用变为计算密集型（CPU bound）程序是很普遍的现象（这意味着此时的瓶颈是CPU而非GPU）。其中的原因是关卡设计师比较偏爱为每个对象用其独有的材质与纹理进行绘制，所以在处理每一个物体时都需要先改变渲染状态再执行绘制调用。当每个API调用都有极高的CPU开销时，为了保证实时的渲染速度，场景的渲染会被限制为只有数千次的绘制调用。图形引擎则会利用批处理技术（batching technique，参见[Wloka03]）来最小化绘制调用的次数。硬件实例化也是如此，有关API将按批处理方式进行高效绘制。

16.1.1 绘制实例数据

说来也许会有些出人意料，在前面各章的演示程序中，我们其实一直都在绘制实例数据！然而，实例的数量却为1（即DrawIndexedInstanced方法的第二个参数）：

```
cmdList->DrawIndexedInstanced(ri->IndexCount, 1,
  ri->StartIndexLocation, ri->BaseVertexLocation, 0);
```

第二个参数InstanceCount指定了所要绘制的几何体实例数量。如

果将此值指定为10，则该几何体将被绘制10次。

照此方法单次调用DrawIndexedInstanced方法来一次性绘制出的10个对象仍不能满足我们的需求，因为这些物体将拥有相同的材质与纹理，且处于相同的位置。因此，下一步就要解决怎样才能为每个实例对象指定它所独有的实例数据，这样的话，我们就能以不同的变换矩阵、材质与纹理渲染出真正有个体差异的实例。

16.1.2　实例数据

在本书的前一版中，实例数据都是自输入装配阶段获取的。在创建输入布局（input layout）时，可以通过枚举项D3D12_INPUT_CLASSIFICATION_PER_INSTANCE_DATA替代D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA来指定输入的数据为逐实例（per-instance）数据流，而非逐顶点（per-vertex）据流。随后，再将第二个顶点缓冲区与含有实例数据的输入流相绑定。[1]Direct3D 12仍然支持这种向流水线传递实例数据的方式，但是我们要另择一种更为现代化的方法。

这所谓的“摩登”方法就是为所有实例都创建一个存有其实例数据的结构化缓冲区。例如，若要将某个对象实例化100次，就应当创建一个具有100个实例数据元素的结构化缓冲区。接着把此结构化缓冲区资源绑定到渲染流水线上，并根据要绘制的实例在顶点着色器中索引相应的数据。那么，怎样才能在顶点着色器中确定要绘制的实例呢？为此，

Direct3D提供了系统值标识符`SV_InstanceID`，可供用户在顶点着色器中方便地实现上述目的。例如，将构成第一个实例所用的各顶点统一编号为0，把组成第二个实例所需的诸顶点统一编号为1，并依此类推。据此，我们便能在顶点着色器中对结构化缓冲区进行索引来获取所需的实例数据。下列着色器代码展示了这一系列的工作流程：

```
// 光源数量的默认值
#ifndef NUM_DIR_LIGHTS
  #define NUM_DIR_LIGHTS 3
#endif

#ifndef NUM_POINT_LIGHTS
  #define NUM_POINT_LIGHTS 0
#endif

#ifndef NUM_SPOT_LIGHTS
  #define NUM_SPOT_LIGHTS 0
#endif

// 包含光照所需的结构体与函数
#include "LightingUtil.hlsl"

struct InstanceData
{
  float4x4 World;
  float4x4 TexTransform;
  uint   MaterialIndex;
  uint   InstPad0;
  uint   InstPad1;
  uint   InstPad2;
};

struct MaterialData
{
  float4  DiffuseAlbedo;
  float3  FresnelR0;
  float  Roughness;
  float4x4 MatTransform;
  uint   DiffuseMapIndex;
  uint   MatPad0;
  uint   MatPad1;
  uint   MatPad2;
```

```
};

// 只有着色器模型5.1+才支持的纹理数组。与Texture2Darray类型数组不同的是，此数组可由不同大小
// 及不同格式的纹理构成，因此它比纹理数组更为灵活
Texture2D gDiffuseMap[7] : register(t0);

// 将下列这两个结构化缓冲区置于space1中，从而避免纹理数组与之重叠而上述纹理数组则将占用t0，
// t1，…，t6寄存器的space0
StructuredBuffer gInstanceData : register(t0, space1);
StructuredBuffer gMaterialData : register(t1, space1);

SamplerState gsamPointWrap    : register(s0);
SamplerState gsamPointClamp    : register(s1);
SamplerState gsamLinearWrap    : register(s2);
SamplerState gsamLinearClamp   : register(s3);
SamplerState gsamAnisotropicWrap : register(s4);
SamplerState gsamAnisotropicClamp : register(s5);

// 每趟绘制过程中都可能会有所变化的常量数据
cbuffer cbPass : register(b0)
{
  float4x4 gView;
  float4x4 gInvView;
  float4x4 gProj;
  float4x4 gInvProj;
  float4x4 gViewProj;
  float4x4 gInvViewProj;
  float3 gEyePosW;
  float cbPerObjectPad1;
  float2 gRenderTargetSize;
  float2 gInvRenderTargetSize;
  float gNearZ;
  float gFarZ;
  float gTotalTime;
  float gDeltaTime;
  float4 gAmbientLight;

  // 对于每个以MaxLights为光源数量最大值的对象来讲，索引[0, NUM_DIR_LIGHTS)表示的是方向光源；
  // 索引[NUM_DIR_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHTS)表示的是点光源；
  // 索引[NUM_DIR_LIGHTS+NUM_POINT_LIGHTS+NUM_SPOT_ LIGHTS]表示的是聚光灯光源

  Light gLights[MaxLights];
};
```

```
struct VertexIn
{
  float3 PosL   : POSITION;
  float3 NormalL : NORMAL;
  float2 TexC   : TEXCOORD;
};

struct VertexOut
{
  float4 PosH   : SV_POSITION;
  float3 PosW   : POSITION;
  float3 NormalW : NORMAL;
  float2 TexC   : TEXCOORD;

  // 由于此处使用的修饰符是nointerpolation，因此该索引指向的都是未经插值的三角形
  nointerpolation uint MatIndex : MATINDEX;
};

VertexOut VS(VertexIn vin, uint instanceID : SV_InstanceID)
{
  VertexOut vout = (VertexOut)0.0f;

  // 获取实例数据
  InstanceData instData = gInstanceData[instanceID];
  float4x4 world = instData.World;
  float4x4 texTransform = instData.TexTransform;
  uint matIndex = instData.MaterialIndex;

  vout.MatIndex = matIndex;

  // 获取材质数据
  MaterialData matData = gMaterialData[matIndex];

  // 将顶点变换到世界空间
  float4 posW = mul(float4(vin.PosL, 1.0f), world);
  vout.PosW = posW.xyz;

  // 假设要执行的是等比缩放，否则就需要使用世界矩阵的逆转置矩阵进行计算
  vout.NormalW = mul(vin.NormalL, (float3x3)world);

  // 把顶点变换到齐次裁剪空间
  vout.PosH = mul(posW, gViewProj);

  // 为了对三角形进行插值而输出顶点属性
  float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), texTransform);
  vout.TexC = mul(texC, matData.MatTransform).xy;
```

```
  return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
  // 获取材质数据
  MaterialData matData = gMaterialData[pin.MatIndex];
  float4 diffuseAlbedo = matData.DiffuseAlbedo;
  float3 fresnelR0 = matData.FresnelR0;
  float roughness = matData.Roughness;
  uint diffuseTexIndex = matData.DiffuseMapIndex;

  // 在数组中动态地查找纹理
  diffuseAlbedo *= gDiffuseMap[diffuseTexIndex].Sample(gsamLinearWrap,
    pin.TexC);

  // 对法线插值可能使它非规范化，因此需要再次对它进行规范化处理
  pin.NormalW = normalize(pin.NormalW);

  // 从表面上一点指向观察点的向量
  float3 toEyeW = normalize(gEyePosW - pin.PosW);

  // 光照项
  float4 ambient = gAmbientLight*diffuseAlbedo;

  const float shininess = 1.0f - roughness;
  Material mat = { diffuseAlbedo, fresnelR0, shininess };
  float3 shadowFactor = 1.0f;
  float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
     pin.NormalW, toEyeW, shadowFactor);

  float4 litColor = ambient + directLight;

  // 从漫反射反照率获取alpha值的常用方法
  litColor.a = diffuseAlbedo.a;

  return litColor;
}
```

可以发现，代码中已不见了物体常量缓冲区（cbPerObject）的身影。此时，每个物体的数据将由实例缓冲区提供。从代码中还能看出，我们是如何通过动态索引来为每个实例关联不同的材质与纹理的。最重要的是，我们可以在单次绘制调用中获取大量的（逐）实例数据！为了

完整性起见，现将上述着色器程序所对应的根签名代码列举如下。

```
CD3DX12_DESCRIPTOR_RANGE texTable;
texTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 7, 0, 0);

// 根参数可以是描述符表、根描述符或根常量
CD3DX12_ROOT_PARAMETER slotRootParameter[4];

// 性能小提示：按变更频率由高到低进行排列，效果更佳
slotRootParameter[0].InitAsShaderResourceView(0, 1);
slotRootParameter[1].InitAsShaderResourceView(1, 1);
slotRootParameter[2].InitAsConstantBufferView(0);
slotRootParameter[3].InitAsDescriptorTable(1, &texTable, D3D12_SHADER_
    VISIBILITY_PIXEL);

auto staticSamplers = GetStaticSamplers();

// 根签名由一系列根参数所组成
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(4, slotRootParameter,
  (UINT)staticSamplers.size(), staticSamplers.data(),
  D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
```

与上一章中所做的工作一样，我们在渲染每一帧画面时都要绑定一次场景中的全部材质与纹理。因而在每次绘制调用时，我们只需再设置存有相应实例数据的结构化缓冲区即可。

```
void InstancingAndCullingApp::Draw(const GameTimer& gt)
{
  ...
  // 绑定此场景所需的全部材质。对于结构化缓冲区而言，我们可以绕过描述符堆的使用而
将其直接设置
  // 为根描述符

  auto matBuffer = mCurrFrameResource->MaterialBuffer->Resource();
  mCommandList->SetGraphicsRootShaderResourceView(1, matBuffer-
    >GetGPUVirtualAddress());

  auto passCB = mCurrFrameResource->PassCB->Resource();
  mCommandList->SetGraphicsRootConstantBufferView(2, passCB-
    >GetGPUVirtualAddress());

  // 绑定渲染此场景所用的一切纹理
```

```
  mCommandList->SetGraphicsRootDescriptorTable(3,
    mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());

  DrawRenderItems(mCommandList.Get(), mOpaqueRitems);
  ...
}

void InstancingAndCullingApp::DrawRenderItems(
  ID3D12GraphicsCommandList* cmdList,
  const std::vector<RenderItem*>& ritems)
{
  // 针对每个渲染项……
  for(size_t i = 0; i < ritems.size(); ++i)
  {
    auto ri = ritems[i];

    cmdList->IASetVertexBuffers(0, 1, &ri->Geo->VertexBufferView());
    cmdList->IASetIndexBuffer(&ri->Geo->IndexBufferView());
    cmdList->IASetPrimitiveTopology(ri->PrimitiveType);

    // 设置此渲染项要用到的实例缓冲区
    // 对于结构化缓冲区来讲，我们可以绕过描述符堆而将其直接设置为根描述符
    auto instanceBuffer = mCurrFrameResource->InstanceBuffer-
    >Resource();
    mCommandList->SetGraphicsRootShaderResourceView(
      0, instanceBuffer->GetGPUVirtualAddress());

    cmdList->DrawIndexedInstanced(ri->IndexCount,
      ri->InstanceCount, ri->StartIndexLocation,
      ri->BaseVertexLocation, 0);
  }
}
```

16.1.3 创建实例缓冲区

实例缓冲区存有绘制每个实例所需的数据，它所存储的信息与之前存于物体常量缓冲区中的数据颇为相似。在CPU端，我们所定义的实例数据结构是这样子的：

```
struct InstanceData
{
```

```
  DirectX::XMFLOAT4X4 World = MathHelper::Identity4x4();
  DirectX::XMFLOAT4X4 TexTransform = MathHelper::Identity4x4();
  UINT MaterialIndex;
  UINT InstancePad0;
  UINT InstancePad1;
  UINT InstancePad2;
};
```

因为渲染项含有实例化次数的相关信息，所以位于系统内存中的实例数据也应算作渲染项结构体的组成部分：

```
struct RenderItem
{
  ...
  std::vector<InstanceData> Instances;
  ...
};
```

为了使GPU可以访问到这些实例数据，还需要用`InstanceData`元素类型创建一个结构化缓冲区。由于该缓冲区是动态缓冲区（即上传缓冲区），所以就能在每一帧都对它进行更新。在例程中，我们仅将可见（visible，指用户视野中可见的物体）实例的实例数据复制到此结构化缓冲区内（这与视锥体剔除有关，详见16.3节），再随着摄像机的各方向移动及四处观察对这些可见实例进行变换。用我们自建的`UploadBuffer`辅助类创建动态缓冲区是极其方便的。

```
struct FrameResource
{
public:

  FrameResource(ID3D12Device* device, UINT passCount,
    UINT maxInstanceCount, UINT materialCount);
  FrameResource(const FrameResource& rhs) = delete;
  FrameResource& operator=(const FrameResource& rhs) = delete;
  ~FrameResource();

  // 在GPU未处理完命令之前，不能对其引用的命令分配器进行重置，因此每一帧都需要有
```

```
它们自己的命令
  // 分配器
  Microsoft::WRL::ComPtr CmdListAlloc;

  // 同理，在GPU未执行完命令之前，也不能对其引用的常量缓冲区进行更新，因此每一帧都需要拥有它们
  // 自己的常量缓冲区
  std::unique_ptr> PassCB = nullptr;
  std::unique_ptr> MaterialBuffer = nullptr;

  // 注意:在此演示程序中，实例只有一个渲染项，所以仅用了一个结构化缓冲区来存储实例数据。若要使程
  // 序更具通用性（即支持实例拥有多个渲染项），我们还需为每个渲染项都添加一个结构化缓冲区，并为每
  // 个缓冲区分配出足够大的空间来容纳要绘制的最多实例个数。虽然看起来涉及的数据有点多，但若不使
  // 用实例化技术，要用到的物体常量数据会更多。例如，如果要在不使用实例化技术的情况下绘制1000
  // 个物体，就必须创建出一个可容纳1000个物体信息的常量缓冲区。但采用了实例化技术后，仅需构建出
  // 能存储1000个实例数据的结构化缓冲区即可

  std::unique_ptr> InstanceBuffer = nullptr;

  // 通过围栏值将命令标记到此围栏点。这使我们可以检测到这些帧资源是否还被GPU所用
  UINT64 Fence = 0;
};

FrameResource::FrameResource(ID3D12Device* device,
  UINT passCount, UINT maxInstanceCount, UINT materialCount)
{
  ThrowIfFailed(device->CreateCommandAllocator(
    D3D12_COMMAND_LIST_TYPE_DIRECT,
    IID_PPV_ARGS(CmdListAlloc.GetAddressOf())));

  PassCB = std::make_unique>(
    device, passCount, true);
  MaterialBuffer = std::make_unique>(
    device, materialCount, false);
  InstanceBuffer = std::make_unique>(
    device, maxInstanceCount, false);
}
```

注意，InstanceBuffer并非常量缓冲区，所以要将最后一个参数指定为false。

16.2　包围体与视锥体

为了实现视锥体剔除（frnstum culling），我们要熟知视锥体与各种包围体（bounding volume，也译作边界盒、外接体等）的数学描述。包围体即近似于目标物体体积的基本几何对象（见图16.1）。尽管包围体只是与物体的形状相近似，但是用数学表示起来比较简单，这使它在工作中更易于使用。

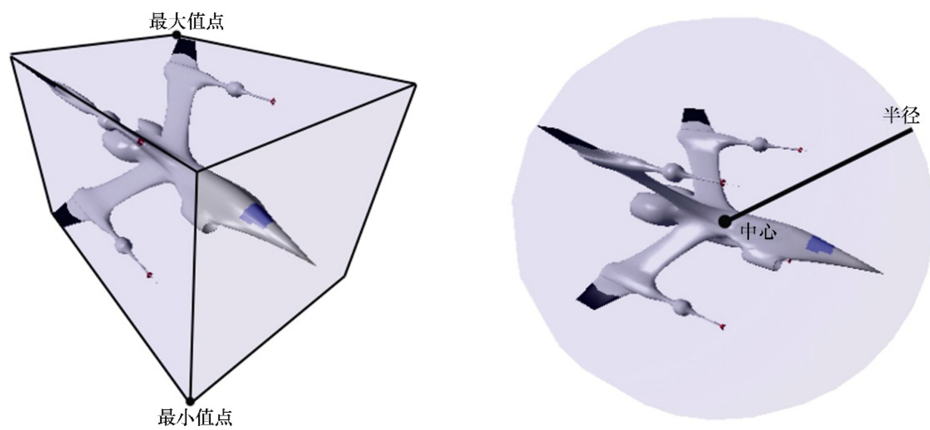


图16.1　分别以轴对齐包围盒与包围球来渲染的网格

16.2.1　DirectXMath碰撞检测库

我们接下来要使用的是DirectXCollision.h工具库，它是DirectXMath库的一个组成部分。此库提供了一份常见几何图元相交测试的快速实现，例如有射线（光）与三角形（ray/triangle）相交检测、射（光）线与（包围）盒（ray/box）相交检测、盒盒（box/box）相交检测、盒与平面（box/plane）相交检测、盒与视锥体（box/frustum）相交检测以及

球体与视锥体（sphere/frustum）相交检测等。在解答本章练习3的过程中，我们将进一步探索此库，并熟悉它所提供的各种功能。

16.2.2 包围盒

网格的轴对齐包围盒（ais-aligned bounding box，AABB）是一种将目标网格紧密包围，且各面皆平行于坐标主轴的长方体。我们可通过最小点$\boldsymbol{v}_{min}$与最大点$\boldsymbol{v}_{max}$来描述AABB（见图16.2）。通过查找目标网格中所有顶点在x、y、z三个坐标轴上所取得的最小值，我们就能得到AABB最小值点$\boldsymbol{v}_{min}$的坐标；反之，遍历目标网格中全部顶点在x、y、z三个坐标轴上可取到的最大值，我们即可求出AABB最大值点$\boldsymbol{v}_{max}$的坐标。

或者以另一种方式来表示AABB：将盒的中心记作$\boldsymbol{c}$、扩展（extents）向量记为$\boldsymbol{e}$，后者存储的是由包围盒中心沿坐标轴至各盒面的距离（见图16.3）。

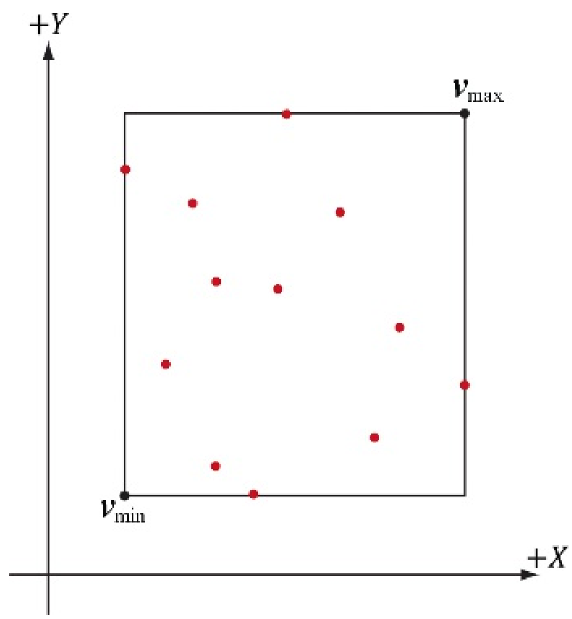


图16.2　用最小值点和最大值点表示包围目标点集的AABB

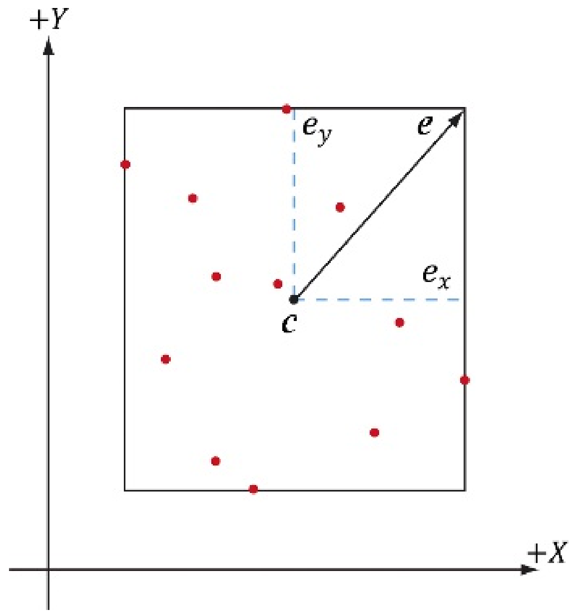


图16.3　用包围盒中心与扩展向量表示包围目标点集的AABB

DirectXMath碰撞检测库采用的是包围盒中心与扩展向量组合的表达方式：

```
struct BoundingBox
{
  static const size_t CORNER_COUNT = 8;

  XMFLOAT3 Center;       // 包围盒的中心
  XMFLOAT3 Extents;      // 中心至各盒面的距离

  ...
```

将上述两种表示法相互转换也是比较方便的。例如，若给出定义包围盒的$\boldsymbol{v}_{min}$与$\boldsymbol{v}_{max}$，则另一种表示法中的包围盒中心及扩展向量可分别表示为

$$\boldsymbol{c} = 0.5(\boldsymbol{v}_{\min} + \boldsymbol{v}_{\max})$$
$$\boldsymbol{e} = 0.5(\boldsymbol{v}_{\max} - \boldsymbol{v}_{\min})$$

下面的代码展示了我们在本章例程中计算骷髅头网格包围盒的具体流程。

```
XMFLOAT3 vMinf3(+MathHelper::Infinity, +MathHelper::Infinity,
    +MathHelper::Infinity);
XMFLOAT3 vMaxf3(-MathHelper::Infinity, -MathHelper::Infinity,
    -MathHelper::Infinity);

XMVECTOR vMin = XMLoadFloat3(&vMinf3);
XMVECTOR vMax = XMLoadFloat3(&vMaxf3);

std::vector<Vertex> vertices(vcount);
for(UINT i = 0; i < vcount; ++i)
{
  fin >> vertices[i].Pos.x >> vertices[i].Pos.y >> vertices[i].Pos.z;
  fin >> vertices[i].Normal.x >> vertices[i].Normal.y >> vertices[i].Norma
l.z;

  XMVECTOR P = XMLoadFloat3(&vertices[i].Pos);

  // 将点投射到单位球面上并生成球面纹理坐标
  XMFLOAT3 spherePos;
  XMStoreFloat3(&spherePos, XMVector3Normalize(P));

  float theta = atan2f(spherePos.z, spherePos.x);

  // 把角度theta限定在 [0, 2pi]区间内
  if(theta < 0.0f)
    theta += XM_2PI;

  float phi = acosf(spherePos.y);

  float u = theta / (2.0f*XM_PI);
  float v = phi / XM_PI;

  vertices[i].TexC = { u, v };

  vMin = XMVectorMin(vMin, P);
  vMax = XMVectorMax(vMax, P);
}

BoundingBox bounds;
XMStoreFloat3(&bounds.Center, 0.5f*(vMin + vMax));
XMStoreFloat3(&bounds.Extents, 0.5f*(vMax - vMin));
```

函数XMVectorMin与XMVectorMax返回的向量分别为

$$\min(\boldsymbol{u},\boldsymbol{v}) = (\min(u_x,v_x),\min(u_y,v_y),\min(u_z,v_z),\min(u_w,v_w))$$
$$\max(\boldsymbol{u},\boldsymbol{v}) = (\max(u_x,v_x),\max(u_y,v_y),\max(u_z,v_z),\max(u_w,v_w))$$

轴对齐包围盒及其旋转操作

图16.4展示了这样一种情况：在某坐标系中的轴对齐包围盒，却没有与其他不同坐标系中的坐标轴相对齐。特别是在局部空间计算出的目标网格的AABB，需要将它进行变换才能得到世界空间中的定向包围盒（oriented bounding box，OBB，也有译作有向包围盒）。[2]但在实际工作过程中，我们通常总是先将网格变换到其局部空间里，再以局部空间内的轴对齐包围盒进行碰撞检测。

另外，我们也能够在世界空间中重新计算网格的AABB，但是有可能得到的是一个与实际物体偏差较大的“肥胖型”长方体（见图16.5）。

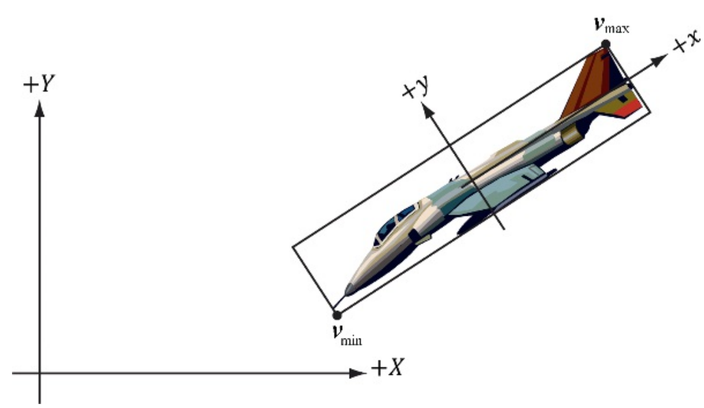


图16.4　包围盒与xy标架的坐标轴相对齐，却没有对齐于XY标架的坐标轴

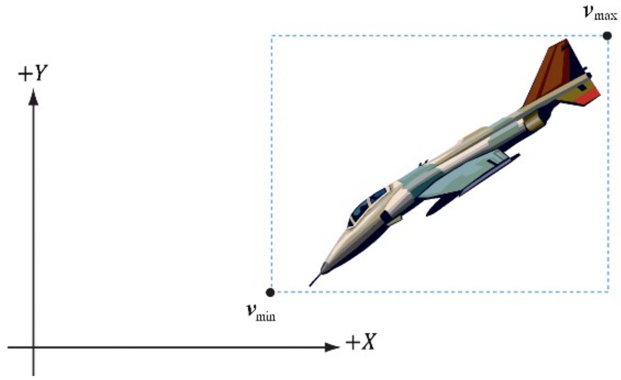


图16.5　轴对齐于XY标架的包围盒

还有一种办法，即放弃轴对齐包围盒，仅采用定向包围盒。此时，我们只需保存好定向包围盒相对于世界空间的朝向即可。DirectX碰撞检测库提供了下述结构体来表示定向包围盒。

```
struct BoundingOrientedBox
{
  static const size_t CORNER_COUNT = 8;

  XMFLOAT3 Center;       // 定向包围盒的中心
  XMFLOAT3 Extents;      // 中心到各面的距离
  XMFLOAT4 Orientation;  // 表示包围盒旋转（box -> world）的单位四元数(unit q
uaternion)
  ...
```

注意 Note

在本章中，我们将看到用四元数（quaternion）表示的旋转/定向操作。简单来讲，一个单位四元数可以像旋转矩阵那样来表示一种旋转动作。我们会在第22章详细讲解四元数这一主题，而现在只要认识到它可以像旋转矩阵那样来表示旋转即可。

通过一个给定的点集并借助DirectX碰撞检测库中的下列静态成员函数，我们就能构建出所需的AABB与OBB：

```
void BoundingBox::CreateFromPoints(
  _Out_ BoundingBox& Out,
  _In_ size_t Count,
  _In_reads_bytes_(sizeof(XMFLOAT3)+Stride*(Count-1)) const XMFLOAT3* pPoi
nts,
  _In_ size_t Stride );

void BoundingOrientedBox::CreateFromPoints(
  _Out_ BoundingOrientedBox& Out,
  _In_ size_t Count,
  _In_reads_bytes_(sizeof(XMFLOAT3)+Stride*(Count-1)) const XMFLOAT3* pPoi
nts,
  _In_ size_t Stride );
```

如果我们定义了顶点结构体如下：

```
struct Basic32
{
  XMFLOAT3 Pos;
  XMFLOAT3 Normal;
  XMFLOAT2 TexC;
};
```

并且，构成网格所用的顶点数组为：

```
std::vector<Vertex::Basic32> vertices;
```

那么，我们就能按下面那样调用函数来生成包围盒：

```
BoundingBox box;
BoundingBox::CreateFromPoints(
  box,
  vertices.size(),
```

```
&vertices[0].Pos,
sizeof(Vertex::Basic32));
```

函数中的Stride（步长）参数表示的是需要越过多少字节才能到达下一个顶点元素处。

注意 Note

为了计算出目标网格的包围体，我们要在系统内存中准备一份可供使用的顶点列表副本，并存在如std::vector这样的类型中。这样做的原因是，CPU无法从以渲染为目的而创建的顶点缓冲区中读取数据。针对这种情况，应用程序中常见的做法就是，为这种数据维护一份存于系统内存中的副本，像拾取（picking，第17章的主题）与碰撞检测（collision detection）两种技术就是这样实现的。

16.2.3 包围球

网格的包围球是一种紧密围绕目标网格的球体。我们可以通过包围球的球心与半径来描述它。一种计算网格包围球的方法是先计算其AABB。接下来再求取AABB的中心，以此作为该包围球的球心：

$$\boldsymbol{c} = 0.5(\boldsymbol{v}_{\min} + \boldsymbol{v}_{\max})$$

包围球的半径事实上是球心c至网格上的任意顶点$\boldsymbol{p}$之间的最大距

离：

$$r = \max\{\|c - p\| : p \in 网格\}$$

假设我们已经计算出了位于局部空间中网格的包围球。经世界变换之后，由于缩放操作的影响，包围球可能已不再紧密围绕于目标网格。这样一来，我们还要对其半径再次进行相应的缩放处理。为了弥补世界变换过程中非等比缩放所带来的影响，我们一定要将包围球的半径按最大缩放分量进行缩放，以便包围球可以完全“包裹”住变换后的网格。另一种可行的策略是，将所有的网格按与游戏场景相同的缩放比例进行建模，由此来避免后续的缩放变换。若使用此方法，只要将模型加载入应用程序即可，而不需要再次对它进行缩放处理。

DirectX碰撞检测库提供了下述结构体来表示包围球：

```
struct BoundingSphere
{
  XMFLOAT3 Center;      // 包围球的球心
  float Radius;         // 包围球的半径
  ...
```

而且，还提供了下列静态成员函数，利用一组点集即可创建包围球：

```
void BoundingSphere::CreateFromPoints(
  _Out_ BoundingSphere& Out,
  _In_ size_t Count,
  _In_reads_bytes_(sizeof(XMFLOAT3)+Stride*(Count-1)) const XMFLOAT3* pPoi
nts,
  _In_ size_t Stride );
```

16.2.4　视锥体

我们在第5章中已经对视锥体有了比较深入的了解，知道了可以在数学上用左、右、顶、底、近、远这6个相交的平面来指定视锥体（平截头体）。现假设这6个视锥体平面都是“内向”朝向的，如图16.6所示。

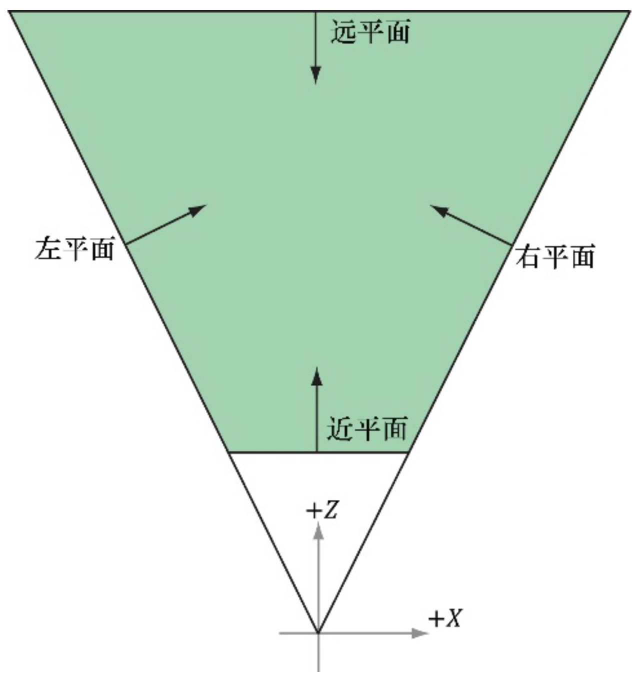


图16.6　视锥体众平面正半空间的交集定义了视锥体的空间范围

这种6个平面的表示法更便于开展视锥体与包围体的相交测试。

16.2.4.1　构建视锥体的众平面

构造视锥体各平面的一种简便方法是：在观察空间中，采取以原点为中点、沿*z*轴正方向俯瞰视锥体的标准形式。此时，可根据在*Z*轴上至原点的距离来方便地确定近平面与远平面，而左平面与右平面、顶平面

与底平面这两组平面则既对称又经过原点（再次观察图16.6）。因此，我们在表达观察空间中的视锥体时，就不必存储所有的平面方程，只需简单地记录顶、底、左、右4个平面的平面斜率，以及近平面与远平面在*Z*轴上至原点的距离即可。DirectX碰撞检测库通过以下结构体来表示视锥体：

```
struct BoundingFrustum
{
  static const size_t CORNER_COUNT = 8;

  XMFLOAT3 Origin;          // 视锥体（及其投影）的原点
  XMFLOAT4 Orientation;     // 表示旋转操作的四元数

  float RightSlope;         // X轴上的正斜率（X/Z），即右平面的斜率
  float LeftSlope;          // X轴上的负斜率，即左平面的斜率
  float TopSlope;           // Y轴上的正斜率（Y/Z），即顶平面的斜率
  float BottomSlope;        // Y轴上的负斜率，即底平面的斜率
  float Near, Far;          // 近平面与远平面在Z轴上至原点的距离
  ...
```

在视锥体的局部空间（例如摄像机的观察空间）中，`Origin`的值为0，且`Orientation`表示的是恒等变换（即不执行任何旋转动作的变换）。我们也可以通过指定`Origin`的位置与四元数`Orientation`的值，用来在世界空间中对视锥体进行定位并改变它的朝向。

若缓存了摄像机视锥体的垂直视场角、纵横比、近平面以及远平面，再辅以一些简单的数学计算，便能确定出观察空间中的视锥体平面方程。当然，通过投影矩阵推导出观察空间中的视锥体平面方程也有多种办法（如[Lengyel02]与[Möuer08]就介绍了两种不同的方案）。DirectXMath碰撞检测库采用下述方式来求取视锥体的平面方程。在NDC（规格化设备坐标）空间中，视锥体便被包围在方盒

$[-1, 1] \times [-1, 1] \times [0, 1]$之内。因此，视锥体的8个角点（corner）可以简化表示为：

```
// 用齐次坐标来表示投影视锥体（projection frustum）中的各角点
static XMVECTORF32 HomogenousPoints[6] =
{
  { 1.0f, 0.0f, 1.0f, 1.0f },   // 计算右平面斜率所用的点(远平面右边中点)
  { -1.0f, 0.0f, 1.0f, 1.0f },  // 计算左平面斜率所用的点(远平面左边中点)
  { 0.0f, 1.0f, 1.0f, 1.0f },   // 计算顶平面斜率所用的点(远平面上边中点)
  { 0.0f, -1.0f, 1.0f, 1.0f },  // 计算底平面斜率所用的点(远平面下边中点)

  { 0.0f, 0.0f, 0.0f, 1.0f },   // 计算近平面到原点距离所用的点(近平面中心点
)
  { 0.0f, 0.0f, 1.0f, 1.0f }    // 计算远平面到原点距离所用的点(远平面中心点
)
};
```

我们能够通过计算投影矩阵的逆矩阵（以及齐次除法的逆运算，可回顾第5章中的有关内容），将NDC空间中的8个角点变换回观察空间。只要求出了观察空间中视锥体的8个角点，我们就能通过一些简单的数学运算来计算出各平面方程（再重申一次，由于在观察空间中视锥体位于原点且对齐于主轴，所以求平面方程的过程不会很复杂）。在DirectX碰撞检测库中，根据投影矩阵计算观察空间中视锥体的代码如下[3]：

```
//-----------------------------------------------------------------------------
// 根据透视投影矩阵来构建视锥体。输入的矩阵中只能含有一个投影。若有旋转、平移或缩放变换，则会构
// 造出不正确的视锥体
//-----------------------------------------------------------------------------
_Use_decl_annotations_
inline void XM_CALLCONV BoundingFrustum::CreateFromMatrix(
  BoundingFrustum& Out,
  FXMMATRIX Projection )
{
```

```
  // 用齐次坐标来表示投影视锥体中的各角点
  static XMVECTORF32 HomogenousPoints[6] =
  {
    { 1.0f, 0.0f, 1.0f, 1.0f },   // 计算右平面斜率所用的点(位于远平面处)
    { -1.0f, 0.0f, 1.0f, 1.0f },  // 计算左平面斜率所用的点
    { 0.0f, 1.0f, 1.0f, 1.0f },   // 计算顶平面斜率所用的点
    { 0.0f, -1.0f, 1.0f, 1.0f },  // 计算底平面斜率所用的点

    { 0.0f, 0.0f, 0.0f, 1.0f },   // 计算近平面到原点距离所用的点
    { 0.0f, 0.0f, 1.0f, 1.0f }    // 计算远平面到原点距离所用的点
  };

  XMVECTOR Determinant;
  XMMATRIX matInverse = XMMatrixInverse( &Determinant, Projection );

  // 计算位于世界空间中的视锥体诸角点
  XMVECTOR Points[6];

  for( size_t i = 0; i < 6; ++i )
  {
    // 把点变换至观察空间
    Points[i] = XMVector4Transform( HomogenousPoints[i], matInverse );
  }

  Out.Origin = XMFLOAT3( 0.0f, 0.0f, 0.0f );
  Out.Orientation = XMFLOAT4( 0.0f, 0.0f, 0.0f, 1.0f );

  // 计算各右、左、顶、底4个平面的斜率
  Points[0] = Points[0] * XMVectorReciprocal( XMVectorSplatZ( Points[0] )
);
  Points[1] = Points[1] * XMVectorReciprocal( XMVectorSplatZ( Points[1] )
);
  Points[2] = Points[2] * XMVectorReciprocal( XMVectorSplatZ( Points[2] )
);
  Points[3] = Points[3] * XMVectorReciprocal( XMVectorSplatZ( Points[3] )
);

  Out.RightSlope = XMVectorGetX( Points[0] );
  Out.LeftSlope = XMVectorGetX( Points[1] );
  Out.TopSlope = XMVectorGetY( Points[2] );
  Out.BottomSlope = XMVectorGetY( Points[3] );

  // 计算近平面与远平面在Z轴上到原点的距离
  Points[4] = Points[4] * XMVectorReciprocal( XMVectorSplatW( Points[4] )
);
  Points[5] = Points[5] * XMVectorReciprocal( XMVectorSplatW( Points[5] )
);
```

```
  Out.Near = XMVectorGetZ( Points[4] );
  Out.Far = XMVectorGetZ( Points[5] );
}
```

16.2.4.2　视锥体与球体的相交检测

对于视锥体剔除来说，我们希望执行的测试之一便是视锥体与球体的相交检测。从中可得知一个球体是否与视锥体相交。注意，如果一个球体完全位于视锥体中则记作相交，这是因为我们将视锥体看作是一种体积，而非边界。由于视锥体是由6个内向（inward facing）平面围成的空间范围，所以它与球体相交的测试过程可以按下列方式执行：如果存在一视锥体平面L，且球体位于L的负半空间（negative half-space）之内，那么我们就判定该球体完全位于视锥体之外。若不存在这样的平面，则此球体与视锥体相交。

因此，视锥体与球体的相交检测就化简为球体与（围成视锥体）平面的6次相交检测。图16.7展示了在球体与平面相交测试的过程中可能会遇到的几种情况。设球体的球心为$\mathbf{c}$且半径为r，则球心至平面的带符号（有向）距离为$k = \mathbf{n} \cdot \mathbf{c} + d$（见附录C）。如果$|k| \leqslant r$，则球体与平面相交；若$k < -r$，那么球体位于平面的后侧；如果$k > r$，则球体位于平面的前侧，且与平面的正半空间相交。鉴于视锥体与球体相交检测的目的，如果球体位于平面的前侧，我们就认为两者是相交的，因为它们相交的位置处于平面所定义的正半空间。

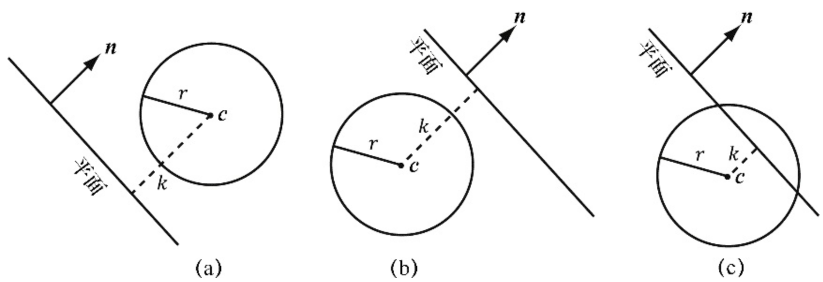


图16.7　球体与平面相交检测可能会遇到的几种情况

（a）$k > r$且球体与平面的正半空间相交

（b）$k < -r$且球体完全位于平面后侧的负半空间之内

（c）$|k| \leqslant r$且球体与平面相交

BoundingFrustum类提供了下列成员函数来测试球体与视锥体是否相交。注意，为了使检测结果有意义，球体与视锥体必须位于同一坐标系内。

```
enum ContainmentType
{
  // 对象完全位于视锥体之外
  DISJOINT = 0,
  // 对象与视锥体的边界相交
  INTERSECTS = 1,
  // 对象完全位于视锥体的空间范围之内
  CONTAINS = 2,
};

ContainmentType BoundingFrustum::Contains(
  _In_ const BoundingSphere& sphere ) const;
```

注意 Note

BoundingSphere类中也相应地含有一个Contains成员函数：

```
ContainmentType BoundingSphere::Contains(
```

```
_In_ const BoundingFrustum& fr ) const;
```

16.2.4.3　视锥体与轴对齐包围盒的相交检测

视锥体与AABB的相交检测和视锥体与球体相交检测所采用的策略相同。由于我们将视锥体的模型定义为6个内向平面围成的体积，所以视锥体与AABB的相交检测可执行如下：如果存在一视锥体平面L，且有包围盒位于L的负半空间，就可推断此盒完全在视锥体之外；若不存在这一平面，我们则判定此包围盒与视锥体相交。

因此，视锥体与AABB的相交检测可化简为6次AABB与平面的相交检测。AABB与平面相交的测试算法如下。先求出穿过包围盒体中心且方向与平面法线$\boldsymbol{n}$最为接近的包围盒体对角向量$\boldsymbol{v} = \overline{PQ}$。根据图16.8所示可知，如果$P$位于平面的前侧，则$Q$也一定位于平面的前侧；若$Q$位于平面的后侧，那么$P$也必定位于平面的后侧；如果$P$位于平面的后侧而$Q$位于平面的前侧，则包围盒与该平面相交。

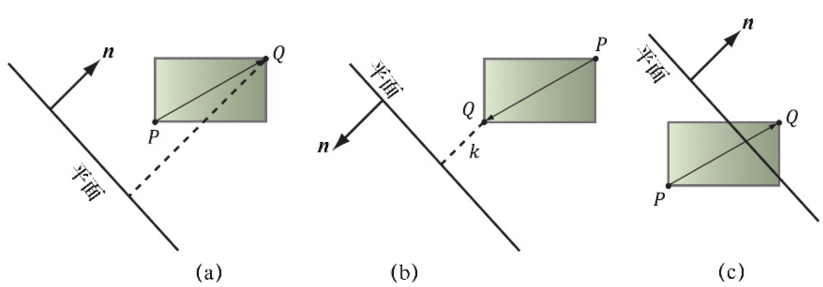


图16.8　AABB与平面相交测试可能会出现的几种情况。$\overline{PQ}$是与平面向量方向最为相近的包围盒体对角线

可以通过下述代码求出方向最接近于平面法向量$\boldsymbol{n}$的PQ：

```
// 针对每个坐标轴x，y，z……
for(int j = 0; j < 3; ++j)
{
  // 找寻在此坐标轴上使PQ与平面法线具有相同方向的点
  if( planeNormal[j] >= 0.0f )
  {
    P[j] = box.minPt[j];
    Q[j] = box.maxPt[j];
  }
  else
  {
    P[j] = box.maxPt[j];
    Q[j] = box.minPt[j];
  }
}
```

这段代码将任务分解为3趟一维空间内的操作，以此选择出使$Q_i - P_i$与平面法线坐标n_i具有相同符号的点P_i与点Q_i（见图16.9）。

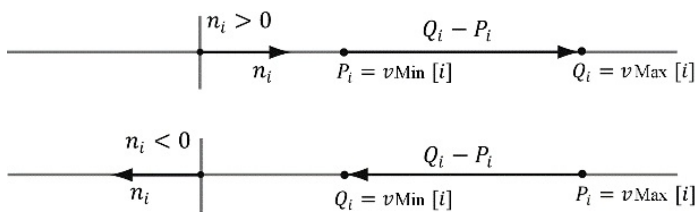


图16.9　上图中，位于第i个坐标轴上的法线分量为正值，因此我们选择$P_i = v\text{Min}[i]$与$Q_i = v\text{Max}[i]$，这样，$Q_i - P_i$的符号便与平面法线坐标n_i相同。下图中，位于第i个坐标轴上的法线分量为负值，所以我们选择$P_i = v\text{Max}[i]$与$Q_i = v\text{Min}[i]$，这样，$Q_i - P_i$便具有与平面法线坐标n_i相同的符号

BoundingFrustum类提供了下列成员函数来测试AABB与视锥体是否相交。注意，为了使测试有意义，务必使AABB与视锥体位于同一坐标系中。

```
ContainmentType BoundingFrustum::Contains(
```

```
  _In_ const BoundingBox& box ) const;
```

注意 Note

BoundingBox类中含有与之对应的成员函数:

```
ContainmentType BoundingBox::Contains(
  _In_ const BoundingFrustum& fr ) const;
```

16.3 视锥体剔除

回顾第5章中所学的内容可知，硬件会在裁剪阶段自动丢弃位于视锥体以外的三角形。但是，当我们拥有数以百万计的三角形时，仍需先通过绘制调用将它们提交至渲染流水线（这会产生API开销），而后再传至顶点着色器，很可能还要经过曲面细分阶段以及几何着色器，直到这些三角形进入裁剪环节才能执行丢弃处理。很明显，该流程的效率是极其低下的。

视锥体裁剪的思路是：利用应用程序代码，在高于以三角形为基本单元（per-triangle basis）的层级中，按组剔除三角形。图16.10展示了一个简单的示例。先来构建包围体，包围盒或包围球都可以，用它们来包围场景中的每一个物体。如果包围体与视锥体不相交，就无须将对应的物体（它可能由上千个三角形构成）交由Direct3D绘制。这样一来，利用开销不大的CPU测试即可节省GPU资源，使它不必在不可见的几何图形上浪费计算时间。假设有一台视场角为90°（垂直方向和水平方向的视场角都是90°，6个同样的观察范围可严丝合缝地覆盖整个场景）且远平面为无穷远的摄像机，该摄像机的视锥体仅占用世界空间的1/6。再假设物体在场景中分布均匀，则世界空间中有5/6的物体会被视锥体剔除方法所丢弃。在实际应用中，摄像机常用小于90°的视场角以及无穷远的远平面，也就是说，会剔除掉场景中5/6以上的物体。

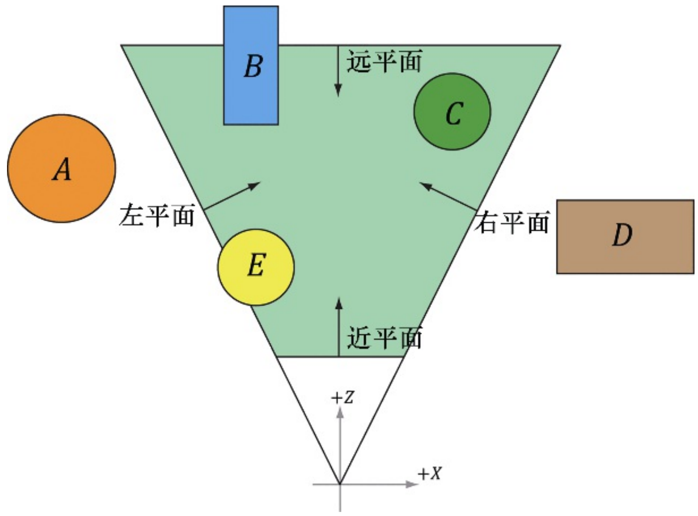


图16.10　被包围体A与D围起来的物体完全位于视锥体之外，所以无须绘制它们。被包围体C围起来的物体体积范围完全位于视锥体以内，所以要对它进行绘制。而被包围体B与E围起来的物体则分为两部分，并分列视锥体内外，因此我们在绘制这两个物体的时候，要通过硬件裁剪丢弃位于视锥体外的那些三角形

在演示程序中，我们要在局部空间中计算骷髅头网格的AABB，并渲染出一个规格为$5 \times 5 \times 5$的骷髅头栅格（见图16.11）。在`UpdateInstanceData`方法里，我们为所有的实例都进行了视锥体裁剪。如果某实例与视锥体相交，就将它加入存有实例数据的结构化缓冲区中的下一个空槽内，并且把计数器`visibleInstanceCount`加1。这样一来，结构化缓冲区前面部分的数据就都是可见的实例了（当然，这个结构化缓冲区的大小与实例的数量相当，为发生所有实例都可见的情况而时刻准备着）。骷髅头网格的AABB位于局部空间，为了执行相交检测，一定要将视锥体变换到每个实例的局部空间中。其实也不必仅拘泥于一种空间的使用，比如说，我们也可将AABB与视锥体一同变换到世界空间。视锥体剔除所更新的代码部分如下：

```
XMMATRIX view = mCamera.GetView();
XMMATRIX invView = XMMatrixInverse(&XMMatrixDeterminant(view), view);
```

```
auto currInstanceBuffer = mCurrFrameResource->InstanceBuffer.get();
for(auto& e : mAllRitems)
{
  const auto& instanceData = e->Instances;

  int visibleInstanceCount = 0;
  for(UINT i = 0; i < (UINT)instanceData.size(); ++i)
  {
    XMMATRIX world = XMLoadFloat4x4(&instanceData[i].World);
    XMMATRIX texTransform = XMLoadFloat4x4(&instanceData[i].TexTransform);

    XMMATRIX invWorld = XMMatrixInverse(&XMMatrixDeterminant(world), world
);

    // 由观察空间到物体局部空间的变换矩阵
    XMMATRIX viewToLocal = XMMatrixMultiply(invView, invWorld);

    // 将摄像机视锥体由观察空间变换到物体的局部空间
    BoundingFrustum localSpaceFrustum;
    mCamFrustum.Transform(localSpaceFrustum, viewToLocal);

    // 在局部空间中执行包围盒与视锥体的相交测试
    if(localSpaceFrustum.Contains(e->Bounds) != DirectX::DISJOINT)
    {
      InstanceData data;
      XMStoreFloat4x4(&data.World, XMMatrixTranspose(world));
      XMStoreFloat4x4(&data.TexTransform, XMMatrixTranspose(texTransform))
;
      data.MaterialIndex = instanceData[i].MaterialIndex;

      // 将可见对象的实例数据写入结构化缓冲区
      currInstanceBuffer->CopyData(visibleInstanceCount++, data);
    }
  }

  e->InstanceCount = visibleInstanceCount;

  // 输出当前可见实例（也就是本帧实际绘制出来的实例数量）的数目与实例的总数作为参
考信息
  std::wostringstream outs;
  outs.precision(6);
  outs << L"Instancing and Culling Demo" <<
    L"  " << e->InstanceCount <<
    L" objects visible out of " << e->Instances.size();
    mMainWndCaption = outs.str();
}
```

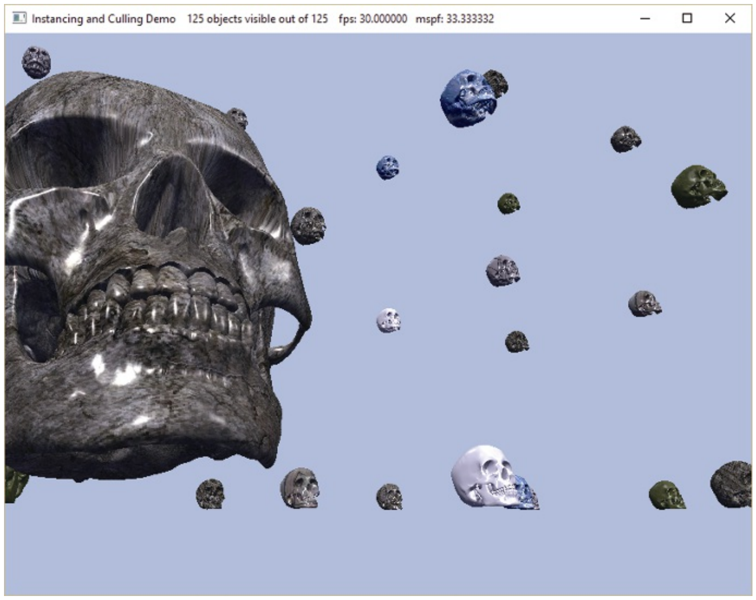


图16.11 “Instancing and Culling”（实例化与视锥体剔除）演示程序的效果

尽管实例缓冲区为每个实例都预留了足够的空间，但只需绘制出编号为0到`visibleInstanceCount-1`所对应的可见实例即可：

```
cmdList->DrawIndexedInstanced(ri->IndexCount,
  ri->InstanceCount,
  ri->StartIndexLocation,
  ri->BaseVertexLocation, 0);
```

图16.12展示了开启与关闭视锥体剔除技术的性能差异。在此示例中，若采用视锥体剔除技术，我们仅向渲染流水线提交13个实例进行绘制，否则我们要为渲染流水线递交所有的125个实例并加以处理。尽管可见的场景部分都是相同的，但若禁用视锥体剔除，我们将白白浪费超过100个骷髅头网格的计算量，而这些数据在裁剪阶段终会被丢掉。每个骷髅头大约由6万个三角形组成，因此会有大量的顶点数据需要处理，并且每个网格都有许多三角形将被抛弃。通过执行一次视锥体与AABB的相交测试，我们就能从即将被送至图形流水线的数据中去掉6

万个本无须处理的三角形——这就是视锥体剔除技术的高明之处。通过比较，我们就能在每秒所绘制的帧数上看出端倪。

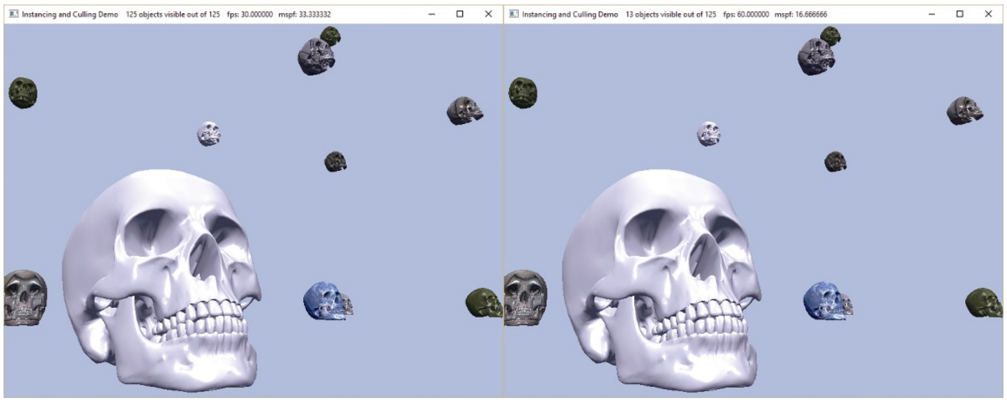


图16.12　在这两幅图里，125个实例中可见的仅占13个。（左图）在视锥体剔除被关闭的情况下，将一共渲染125个实例，渲染一帧要花费33.33毫秒。（右图）在开启视锥体剔除的情况下，帧率加倍

16.4　小结

1．实例化技术可用于将场景中的同一对象，以不同位置、朝向、缩放（大小）、材质与纹理等绘制多次。为了节约内存资源，我们可以仅创建一个网格，再利用不同的世界矩阵、材质以及纹理向Direct3D提交多个绘制调用。为了避免资源变动和多次绘制调用所带来的API开销，我们可以给存有全部实例数据的结构化缓冲区绑定一个SRV，并利用`SV_InstanceID`系统值在顶点着色器中对其进行索引。另外，我们还可以通过动态索引来索引纹理数组。单次绘制调用中要渲染的实例个数由`ID3D12GraphicsCommandList::DrawIndexedInstanced`方法的第二参数`InstanceCount`指定。

2．包围体是近似于目标物体体积的基本几何对象。尽管包围体仅与目标物体的形状相似，但它的数学描述却比较简单，这使得它在工作过程中更便于使用。常见的包围体有包围球、轴对齐包围盒（AABB）以及定向包围盒（OBB）等。碰撞检测库的DirectXCollision.h头文件中定义了表示各种包围体的结构体、对它们进行变换的多种函数以及多种相交测试方法。

3．GPU会在裁剪阶段自动抛弃位于视锥体之外的三角形。但是，那些终将被裁剪的三角形仍要先通过绘制调用（会产生API开销）提交至渲染流水线，并经过顶点着色器的处理，还极有可能传到曲面细分阶段与几何着色器内，直到在裁剪阶段中才能被丢弃。为了改善这种无效率的处理流程，我们可以采用视锥体剔除技术。此方法的思路是构建一

个包围体，包围球、包围盒都可以，使它们分别包围场景中的每一个物体。如果包围体与视锥体没有交集，则无须将物体交给Direct3D绘制。此法通过开销较小的CPU测试大大节省了GPU资源的浪费，从而使它不必为看不到的几何图形“买单”。

16.5　练习

1．修改“Instancing and Culling”（实例化与视锥体裁剪）演示程序：用包围球代替其中的包围盒。

2．在NDC空间中，平面方程的形式异常简单。视锥体内的所有点都被约束在以下的空间范围内：

$$
\begin{aligned}
-1 \leqslant x_{ndc} \leqslant 1 \\
-1 \leqslant y_{ndc} \leqslant 1 \\
0 \leqslant z_{ndc} \leqslant 1
\end{aligned}
$$

特别是NDC空间中视锥体的左平面方程与右平面方程，分别为$x=-1$与$x=1$。而在进行透视除法之前的齐次裁剪空间中，视锥体里的所有点则被约束在下列范围之内：

$$
\begin{aligned}
-w \leqslant x_h \leqslant w \\
-w \leqslant y_h \leqslant w \\
0 \leqslant z_h \leqslant w
\end{aligned}
$$

此时，视锥体的左平面被定义为$w=-x_h$，而右平面的定义为$w=x_h$。设$\boldsymbol{M}=\boldsymbol{VP}$为观察矩阵与投影矩阵的乘积，且$\boldsymbol{v}=(x,\ y,\ z,\ 1)$为世界空间中视锥体内的一点。思考并根据$(x_h,\ y_h,\ z_h,\ w)=\boldsymbol{vM}=(\boldsymbol{v}\cdot\boldsymbol{M}_{*,1},\boldsymbol{v}\cdot\boldsymbol{M}_{*,2},\boldsymbol{v}\cdot\boldsymbol{M}_{*,3},\boldsymbol{v}\cdot\boldsymbol{M}_{*,4})$来证明位于世界空间中视锥体的6个内向平面可分别表示为：

左平面	$0=\boldsymbol{v}\cdot(\boldsymbol{M}_{*,1}+\boldsymbol{M}_{*,4})$

右平面	$0=\boldsymbol{v}\cdot(\boldsymbol{M}_{*,4}-\boldsymbol{M}_{*,1})$
底平面	$0=\boldsymbol{v}\cdot(\boldsymbol{M}_{*,2}+\boldsymbol{M}_{*,4})$
顶平面	$0=\boldsymbol{v}\cdot(\boldsymbol{M}_{*,4}-\boldsymbol{M}_{*,2})$
近平面	$0=\boldsymbol{v}\cdot\boldsymbol{M}_{*,3}$
远平面	$0=\boldsymbol{v}\cdot(\boldsymbol{M}_{*,4}-\boldsymbol{M}_{*,3})$

注意 Note

（a）题目中的平面法线皆指向视锥体的内部。这就意味着由6个边界平面到视锥体内部某点的距离都为正值。换句话说，对于视锥体内的任意一点$\boldsymbol{p}$，有$\boldsymbol{n}\cdot\boldsymbol{p}+d\geqslant 0$。

（b）可以观察到$v_w=1$，因此上述点积公式均可代以形如$Ax+By+Cz+D=0$的平面方程。

（c）若计算出的平面法向量并不是单位长度，则可参考附录C中如何对一个平面进行规范化处理。

3．考察DirectXCollision.h头文件，研究它为相交检测与包围体变换所提供的相关函数。

4．定义一个OBB所需的条件有：一个中心点$\boldsymbol{C}$，3个用于定义OOB朝向的相互正交的轴向量$\boldsymbol{r}_0$、$\boldsymbol{r}_1$、$\boldsymbol{r}_2$，以及3个分别在OOB轴$\boldsymbol{r}_0$、$\boldsymbol{r}_1$、$\boldsymbol{r}_2$方向上的扩展（extent）长度$\boldsymbol{a}_0$、$\boldsymbol{a}_1$和$\boldsymbol{a}_2$，借此给出从OOB中心至其各面的3种距离长度。

（a）观察图16.13（图中展示的是2D场景），证明OBB投影在法向量$\boldsymbol{n}$所定义的轴上的“阴影”长度为$2r$，其中

$$r = |a_0\boldsymbol{r}_0 \cdot \boldsymbol{n}| + |a_1\boldsymbol{r}_1 \cdot \boldsymbol{n}| + |a_2\boldsymbol{r}_2 \cdot \boldsymbol{n}|$$

（b）解释：上述求取r的公式中为什么一定要用绝对值，而不是用$r = (a_0\boldsymbol{r}_0 + a_1\boldsymbol{r}_1 + a_2\boldsymbol{r}_2) \cdot \boldsymbol{n}$来计算其值。

（c）推导出平面与OBB相交检测的方法，用来分别确定OBB位于平面之前、平面之后以及与平面相交的这几种情况。

（d）AABB是OBB的一个特例，因此OBB的相交检测也适用于AABB。但是，与AABB所对应的求r公式是可以进一步化简的。试推导出针对AABB简化后的求r公式。

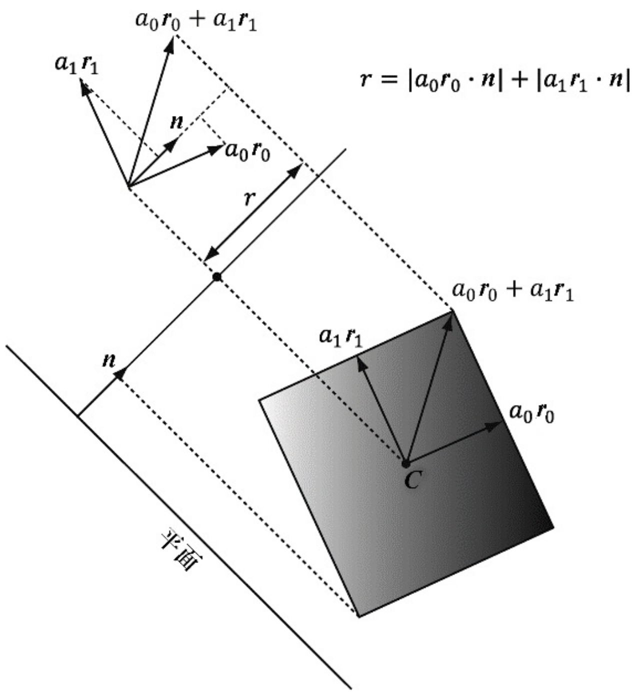


图16.13　平面与OBB相交检测的示意图

---

[1]　此“旧办法”的思路可参见《Efficiently Drawing Multiple Instances of Geometry》(bb173349)一文或本书前一版本。

[2]　为区别于AABB这种特例而称的包围盒，也就是任意朝向的包围盒。位于物体局部空间的OBB也称为OOBB，即object-oriented bounding box。

[3]　DirectMath库不定期调整，因此其下的碰撞检测库也是如此，最新版与文中代码细节可能稍有出入。

第17章　拾取

本章要讨论的问题是如何来确定用户通过鼠标指针所拾取的3D物体（或图元）（见图17.1）。换言之，若给出鼠标指针点击的2D屏幕坐标，是否能够确定投影到此点上的3D对象呢？就此而言，为了解决这个问题，在某种意义上来说，我们必须要做一些“逆于常规”的事情。即往常我们会将物体从3D空间变换到屏幕空间，但此时却要将物体从屏幕空间变换回3D空间。话虽简单，然而在此过程中还是会遇到一点儿麻烦：即，对应于特定2D屏幕点的3D点并不是唯一的（可能会存在多个3D点投影到2D投影窗口中同一点上的情况，见图17.2）。这样一来，在确定拾取对象的过程中还存在着一丝不明确之处。然而这并不是非常严重的问题，因为用户选取的物体往往是距离摄像机最近的那一个。

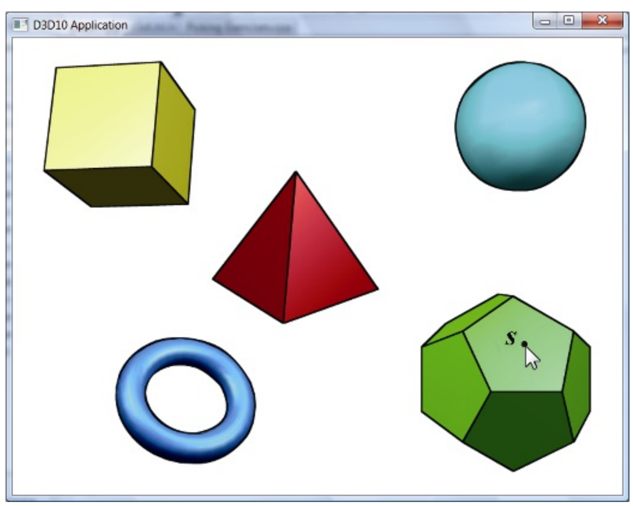


图17.1　用户正在选取多个物体之中的十二面体

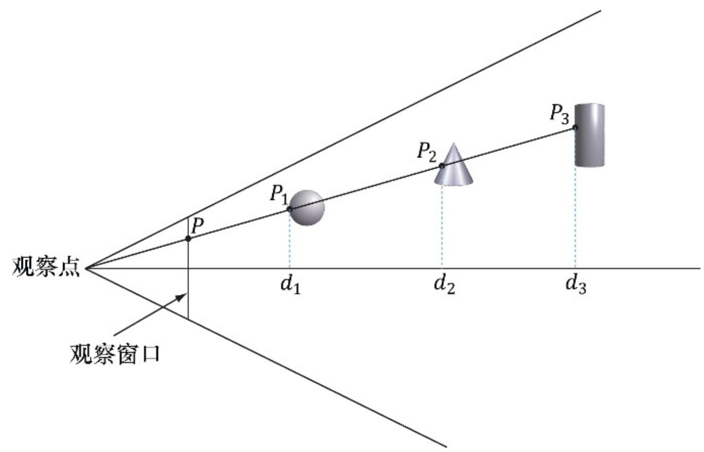


图17.2　视锥体的侧视图。不难看出，3D空间中有多个点都可以投影到投影窗口内的同一个点上

考虑图17.3，该图中展示了这样一个视锥体。其中，位于投影窗口中的点$\boldsymbol{p}$对应于用户在屏幕上单击的点$\boldsymbol{s}$。观察此图后可知，如果以观察点为起点，发出一条穿过点$\boldsymbol{p}$的拾取射线（picking ray），则此射线会与投影中含有点$\boldsymbol{p}$的物体相交，在此示例中的相交物体是一个圆柱体。据此，拾取方案如下：一旦计算出了拾取射线，我们就能遍历场景中的每一个物体，并检测此射线是否与它相交。而与射线相交的物体就是用户所拾取的对象。正如之前所提到的，射线可能会与场景中的多个物体相交（也可能一个相交的都没有，即用户什么都没选），但是位于射线路径之上的物体会具有不同的深度值。在这种情况下，我们就选择与射线相交且距摄像机最近的物体作为拾取对象。

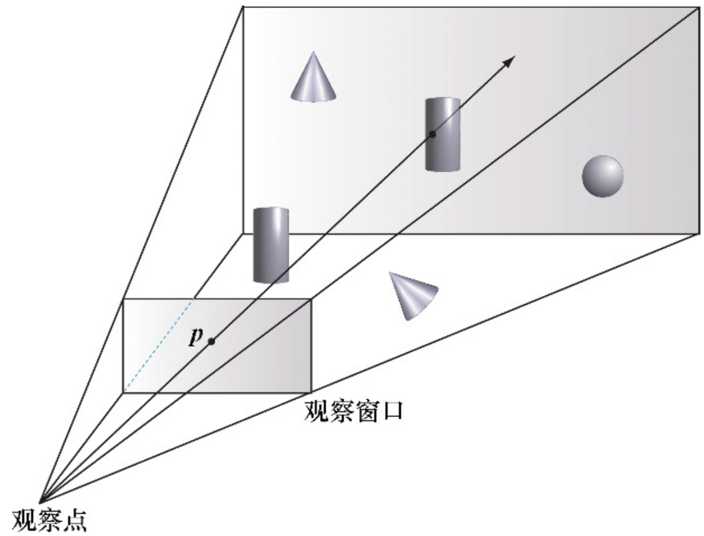


图17.3　从观察点发出的一条经点$\boldsymbol{p}$的射线会交于投影中含有点$\boldsymbol{p}$的物体。注意，位于投影窗口中的投影点$\boldsymbol{p}$对应于用户在屏幕上点选的点$\boldsymbol{s}$

学习目标：

为了学习如何实现拾取算法并理解它的工作原理，我们将拾取过程分为4个步骤。

（a）根据用户在屏幕上点选的点$\boldsymbol{s}$，求出在投影窗口中与之对应的点$\boldsymbol{p}$。

（b）计算出位于观察空间中的拾取射线。此射线以观察空间中的原点作为起点，并经过点$\boldsymbol{p}$。

（c）将拾取射线与场景中用于相交检测的模型变换到同一空间之中。

（d）确定与拾取射线相交的物体。与射线相交且距摄像机最近的物体即为用户在屏幕中拾取的对象。

17.1 屏幕空间到投影窗口的变换

我们的首要任务就是将用户在屏幕中点选的点变换为规格化设备坐标（Normalized Device Coordinates，NDC参见5.6.3.3节）。重温用视口矩阵（viewport matrix）将顶点从规格化设备坐标变换到屏幕空间的过程，视口矩阵如下：

$$\boldsymbol{M}=\begin{bmatrix}\frac{\text{Width}}{2} & 0 & 0 & 0\\ 0 & -\frac{\text{Height}}{2} & 0 & 0\\ 0 & 0 & \text{MaxDepth}-\text{MinDepth} & 0\\ \text{TopLeftX}+\frac{\text{Width}}{2} & \text{TopLeftY}+\frac{\text{Height}}{2} & \text{MinDepth} & 1\end{bmatrix}$$

视口矩阵中的变量可通过D3D12_VIEWPORT结构体来设置：

```
typedef struct D3D12_VIEWPORT
{
  FLOAT TopLeftX;
  FLOAT TopLeftY;
  FLOAT Width;
  FLOAT Height;
  FLOAT MinDepth;
  FLOAT MaxDepth;
} D3D12_VIEWPORT;
```

对于游戏来讲，视口的大小往往是整个后台缓冲区的尺寸，而深度缓冲区的范围则为0～1。如此一来，视口矩阵中所用的变量将分别为$TopLeftX=0$、$TopLeftY=0$、$MinDepth=0$、$MaxDepth=1$、$Width=w$以及$Height=h$，其中，w与h分别为后台缓冲区的宽度与高度。假设当前的设置方案正是如此，则视口矩阵可化简为：

$$\boldsymbol{M}=\begin{bmatrix} w/2 & 0 & 0 & 0 \\ 0 & -h/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ w/2 & h/2 & 0 & 1 \end{bmatrix}$$

现设$\boldsymbol{p}_{ndc}=(x_{ndc},\ y_{ndc},\ z_{ndc},\ 1)$为规格化设备空间（即$-1\leqslant x_{ndc}\leqslant 1$，$-1\leqslant y_{ndc}\leqslant 1$与$0\leqslant z_{ndc}\leqslant 1$所围成的空间）中的一点。将$\boldsymbol{p}_{ndc}$变换到屏幕空间：

$$[x_{ndc},\ y_{ndc},\ z_{ndc},\ 1]\begin{bmatrix} w/2 & 0 & 0 & 0 \\ 0 & -h/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ w/2 & h/2 & 0 & 1 \end{bmatrix}=\left[\frac{x_{ndc}w+w}{2},\ \frac{-y_{ndc}h+h}{2},\ z_{ndc},\ 1\right]$$

坐标z_{ndc}仅被深度缓冲区所用，而我们在讨论拾取技术时并不涉及任何与深度有关的坐标。因此，仅需对$\boldsymbol{p}_{ndc}$的x坐标与y坐标进行变换，即可求出它在2D屏幕空间中所对应的点$\boldsymbol{p}_s=(x_s,y_s)$：

$$x_s=\frac{x_{ndc}w+w}{2}$$
$$y_s=\frac{-y_{ndc}h+h}{2}$$

给定视口的大小以及规格化设备坐标系中的点$\boldsymbol{p}_{ndc}$，我们就能根据上述方程求出屏幕空间里的点$\boldsymbol{p}_s$。然而，在实际的拾取情景中，我们却要通过已知的屏幕上一点$\boldsymbol{p}_s$以及视口的大小，求出点$\boldsymbol{p}_{ndc}$。对上述公式变形即可得到求取点$\boldsymbol{p}_{ndc}$的方程：

$$x_{ndc}=\frac{2x_s}{w}-1$$
$$y_{ndc}=-\frac{2y_s}{h}+1$$

现在便得到了用户选取的点在NDC空间中的位置。但是，要找到对应的拾取射线，我们还需求出此点位于观察空间中的屏幕点。在5.6.3.3节中，我们是通过令x坐标除以纵横比r来将投影点从观察空间变换到NDC空间的：

$$-r \leqslant x' \leqslant r$$
$$-1 \leqslant \frac{x}{r} \leqslant 1$$

因此，若要再变换回观察空间，仅需为NDC空间中的x坐标乘以纵横比即可。用户单击的点在观察空间中坐标则为：

$$x_v = r\left(\frac{2s_x}{w} - 1\right)$$
$$y_v = -\frac{2s_y}{h} + 1$$

注意 Note

由于我们将观察空间中投影窗口的高度区间定为[−1, 1]，因此投射到观察空间的y坐标与NDC空间中y坐标是相同的。

在5.6.3.1节中，我们定义投影窗口位于距原点$d = \cot\left(\frac{\alpha}{2}\right)$处，这里的$\alpha$为垂直视场角。由此，我们就能发出经过投影窗口上的点(x_v, y_v, d)的拾取射线。然而，若采用这种方法，我们还需计算$d = \cot\left(\frac{\alpha}{2}\right)$。对此，

这里给出另一种更简单的处理方式，如图17.4所示。

$$x_v^{'} = \frac{x_v}{d} = \frac{x_v}{\cot\left(\frac{\alpha}{2}\right)} = x_v \cdot \tan\left(\frac{\alpha}{2}\right) = \left(\frac{2s_x}{w} - 1\right) r \tan\left(\frac{\alpha}{2}\right)$$

$$y_v^{'} = \frac{y_v}{d} = \frac{y_v}{\cot\left(\frac{\alpha}{2}\right)} = y_v \cdot \tan\left(\frac{\alpha}{2}\right) = \left(-\frac{2s_y}{h} + 1\right) \tan\left(\frac{\alpha}{2}\right)$$

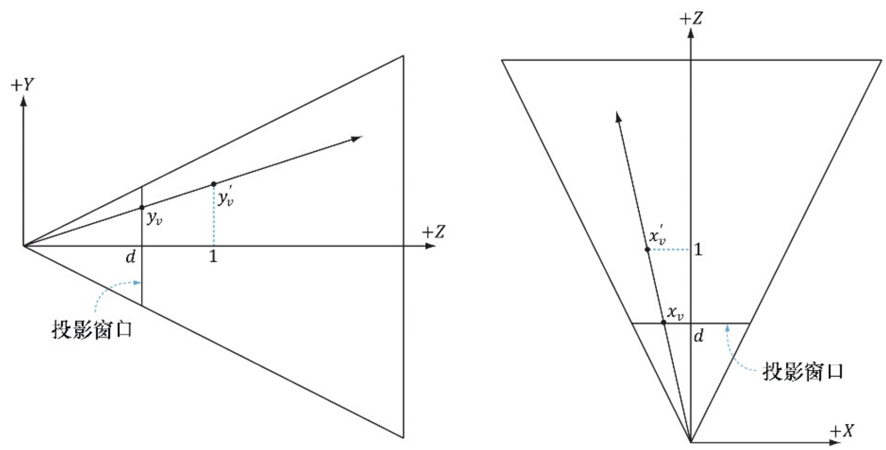


图17.4　根据相似三角形可知，$\frac{y_v}{d} = \frac{y_v^{'}}{1}$以及$\frac{x_v}{d} = \frac{x_v^{'}}{1}$

又由于在投影矩阵中，$\boldsymbol{P}_{00} = \frac{1}{r\tan\left(\frac{\alpha}{2}\right)}$且$\boldsymbol{P}_{11} = \frac{1}{\tan\left(\frac{\alpha}{2}\right)}$（见5.6.3.4节），我们便可将它们改写为：

$$x_v^{'} = \frac{\left(\frac{2s_x}{w} - 1\right)}{\boldsymbol{P}_{00}}$$

$$y_v^{'} = \frac{\left(-\frac{2s_y}{h} + 1\right)}{\boldsymbol{P}_{11}}$$

这样一来，我们就可以令拾取射线改为穿过点$(x_v^{'}, y_v^{'}, 1)$，而此射线与通过点(x_v, y_v, d)的其实也是同一条拾取射线。计算观察空间中拾取射线的代码如下：

```
void PickingApp::Pick(int sx, int sy)
{
  XMFLOAT4X4 P = mCamera.GetProj4x4f();

  // 计算观察空间中的拾取射线
  float vx = (+2.0f*sx / mClientWidth - 1.0f) / P(0, 0);
  float vy = (-2.0f*sy / mClientHeight + 1.0f) / P(1, 1);

  // 位于观察空间中拾取射线的定义
  XMVECTOR rayOrigin = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
  XMVECTOR rayDir = XMVectorSet(vx, vy, 1.0f, 0.0f);
```

注意，拾取射线的端点（也称为拾取射线的原点）位于观察空间的原点，这是因为观察点就在观察空间的原点处。

17.2 位于世界空间与局部空间中的拾取射线

我们刚刚获得了观察空间中的拾取射线，但是它只可用于处理观察空间中的物体。由于利用观察矩阵能将几何体从世界空间变换到观察空间，因此，借助观察矩阵的逆矩阵便会使几何体由观察空间变换回世界空间。如果$\boldsymbol{r}_v(t)=\boldsymbol{q}+t\boldsymbol{u}$为观察空间中的拾取射线，且$\boldsymbol{V}$是观察矩阵，那么世界空间中的拾取射线则为：

$$\begin{aligned}\boldsymbol{r}_w(t)&=\boldsymbol{q}\boldsymbol{V}^{-1}+t\boldsymbol{u}\boldsymbol{V}^{-1}\\&=\boldsymbol{q}_w+t\boldsymbol{u}_w\end{aligned}$$

注意，射线的端点$\boldsymbol{q}$变换后是一个点（即$q_w=1$），而射线方向$\boldsymbol{u}$变换后为一个向量（即$u_w=0$）。

对于在世界空间中物体的情景来说，世界空间拾取射线十分有用。但是在大多数情况下，物体的几何图形往往是相对于其局部空间来定义的。因此，为了执行拾取射线与物体的相交检测，我们一定要将射线变换到物体所在的局部空间中。如果$\boldsymbol{W}$是物体的世界矩阵，那么通过矩阵$\boldsymbol{W}^{-1}$便可将几何体从世界空间变换到此物体的局部空间。因此，局部空间中的拾取射线则为：

$$\boldsymbol{r}_L(t)=\boldsymbol{q}_w\boldsymbol{W}^{-1}+t\boldsymbol{u}_w\boldsymbol{W}^{-1}$$

一般来讲，场景中的物体都有自己的局部空间。因此，一定要将拾取射线变换到场景中每个物体所在的局部空间，才能与之开展相交检测。

有读者能会说，不如把网格变换到世界空间再执行相交测试。然而，这种做法的代价实在是太昂贵了：一个网格就可能含有上千个顶点，而我们需要将所有这些顶点都变换到世界空间。很明显，还是把拾取射线变换到物体所在局部空间中的做法效率更高。

下列代码展示了如何将拾取射线从观察空间变换到物体的局部空间：

```
// 假设在开始时用户并没有拾取任何点，因此将拾取渲染项设置为不可见
mPickedRitem->Visible = false;

// 检测用户是否拾取了一个不透明的渲染项
// 实际的应用程序可能会单独维护一个由可选物体组成的“拾取列表”
for(auto ri : mRitemLayer[(int)RenderLayer::Opaque])
{
  auto geo = ri->Geo;

  // 跳过不可见的渲染项
  if(ri->Visible == false)
    continue;

  XMMATRIX V = mCamera.GetView();
  XMMATRIX invView = XMMatrixInverse(&XMMatrixDeterminant(V), V);

  XMMATRIX W = XMLoadFloat4x4(&ri->World);
  XMMATRIX invWorld = XMMatrixInverse(&XMMatrixDeterminant(W), W);

  // 将拾取射线变换到网格局部空间
  XMMATRIX toLocal = XMMatrixMultiply(invView, invWorld);

  rayOrigin = XMVector3TransformCoord(rayOrigin, toLocal);
  rayDir = XMVector3TransformNormal(rayDir, toLocal);

  // 为相交检测而计算拾取射线方向上的单位长度
  rayDir = XMVector3Normalize(rayDir);
```

函数XMVector3TransformCoord与XMVector3TransformNormal都以3D向量作为参数，还需注意的是，XMVector3TransformCoord函

数总是把参数向量的第四个分量认作$w = 1$（即返回$w = 1$的向量），而XMVector3TransformNormal函数则会将参数向量的第四个分量看作$w = 0$（即返回$w = 0$的向量）。因此，我们就能通过XMVector3TransformCoord函数来变换点，而用XMVector3TransformNormal函数对向量进行变换。

17.3 射线与网格的相交检测

拾取射线一旦与网格位于同一空间，我们就能通过相交检测来验证这两者是否相交。以下代码的功能是遍历网格中的所有三角形，并一一执行射线与三角形的相交检测。如果射线与其中的一个三角形相交，说明该射线一定射中了此三角形所在的网格，否则，此射线则与该网格无缘。一般来讲，我们只关心与射线相交且距摄像机最近的三角形，因为若有多个三角形叠在射线所经过的路径上，则会有多个三角形与射线相交。

```
// 如果拾取射线碰到了网格的包围盒，那么用户就有可能拾取了网格上的一个三角形。因此
，我们就应进一
// 步执行射线与三角形的相交测试
// 如果拾取射线没有碰到网格的包围盒，说明用户也一定没有点击到此网格，因此也就不必
进行射线与该网
// 格上的三角形相交检测这无用功
float tmin = 0.0f;
if(ri->Bounds.Intersects(rayOrigin, rayDir, tmin))
{
  // 注意：对于此演示程序来说，我们是知道如何强制转换顶点与索引的数据格式的。但对
于不同格式混合
  // 在一起的情况而言，则需要用元数据（metadata）来执行强制类型转换
  auto vertices == (Vertex*)geo->VertexBufferCPU->GetBufferPointer();
  auto indices == (std::uint32_t*)geo->IndexBufferCPU->GetBufferPointer()
;
  UINT triCount = ri->IndexCount / 3;

  // 对找到的离摄像机最近的三角形执行它与（拾取）射线的相交检测
  tmin = MathHelper::Infinity;
  for(UINT i = 0; i < triCount; ++i)
  {
    // 此三角形的索引
    UINT i0 = indices[i * 3 + 0];
    UINT i1 = indices[i * 3 + 1];
    UINT i2 = indices[i * 3 + 2];

    // 构成此三角形的顶点
```

```
    XMVECTOR v0 = XMLoadFloat3(&vertices[i0].Pos);
    XMVECTOR v1 = XMLoadFloat3(&vertices[i1].Pos);
    XMVECTOR v2 = XMLoadFloat3(&vertices[i2].Pos);

    // 为了找到距摄像机最近的与拾取射线相交的三角形，我们必须遍历网格上的所有三角
形
    float t = 0.0f;
    if(TriangleTests::Intersects(rayOrigin, rayDir, v0, v1, v2, t))
    {
      if(t < tmin)
      {
        // 这是目前距离摄像机最近的一个被拾取的三角形
        tmin = t;
        UINT pickedTriangle = i;

        // 为被拾取的三角形设置渲染项，使我们可以用特定的“highlight”（高亮突出
）材质来对它
        // 进行渲染
        mPickedRitem->Visible = true;
        mPickedRitem->IndexCount = 3;
        mPickedRitem->BaseVertexLocation = 0;

        // 被拾取的渲染项需要与被拾取的物体使用相同的世界矩阵
        mPickedRitem->World = ri->World;
        mPickedRitem->NumFramesDirty = gNumFrameResources;

        // 偏移到被拾取三角形在网格索引缓冲区中的索引处
        mPickedRitem->StartIndexLocation = 3 * pickedTriangle;
      }
    }
  }
}
```

可以发现，在处理拾取操作的过程中，我们使用的是MeshGeometry类中存储的网格几何体在系统内存中的副本。这样做的原因是我们无法以读取数据的方式来访问GPU即将绘制的顶点缓冲区及索引缓冲区。因此，在使用拾取与碰撞检测这样的技术时，我们常常会在系统内存中存储一份几何体的副本。有时候，我们又会以节约内存与计算能力为目的来存储一些精简化的网格。

17.3.1　射线与轴对齐包围盒的相交检测

考察我们使用的第一个DirectX碰撞检测库函数，即用于判定射线与网格包围盒是否相交的BoundingBox::Intersects函数。说起来，这与我们在前一章中谈及的视锥体剔除优化技术比较相似。对场景中的每个三角形逐一执行与射线的相交检测会使运算时间大大增加。此时，甚至包括远离拾取射线的网格上的三角形在内，都要一一遍历，以确认这些网格与射线完全没有交集，整个过程极其粗暴且低效。对此，一种常见的做法是采用一个接近于网格的简单包围体，如包围球或包围盒。接下来，我们首先以射线与包围体的相交检测来取代射线与网格的相交测试。如果射线与包围体没有交集，则该射线必然避开了此三角形网格，因而也就无需进行后续的计算了。但若射线与包围体有交集，那么我们就要执行更精确的射线与网格的相交检测。假使射线错过了场景中大多数的包围体，便会节省我们不少次的射线与三角形相交检测。如果射线与包围盒相交，则BoundingBox::Intersects函数返回true，否则返回false。该函数的原型为：

```
bool XM_CALLCONV
BoundingBox::Intersects(
  FXMVECTOR Origin,    // 射线的原点（端点）
  FXMVECTOR Direction, // 射线的方向向量（必为单位长度）
  float& Dist ); const // 射线的相交参数
```

给定射线$\boldsymbol{r}(t)=\boldsymbol{q}+t\boldsymbol{u}$，则最后一个参数输出的是实际相交点$\boldsymbol{p}$的射线参数$t_0$：

$$\boldsymbol{p}=\boldsymbol{r}(t_0)=\boldsymbol{q}+t_0\boldsymbol{u}$$

17.3.2　射线与球体的相交检测

DirectX碰撞检测库还提供了一个射线与球体的相交检测函数：

```
bool XM_CALLCONV
BoundingSphere::Intersects(
  FXMVECTOR Origin,
  FXMVECTOR Direction,
  float& Dist ); const
```

为了对这些检测有更深入的理解，我们将展示射线与球体相交测试的推导过程。球心为$\boldsymbol{c}$、半径为r的球体上一点$\boldsymbol{p}$满足方程：

$$||\boldsymbol{p} - \mathbf{c}|| = r$$

设$\boldsymbol{r}(t) = \boldsymbol{q} + t\boldsymbol{u}$为一条射线。我们希望能解得在取$t_1$与$t_2$时，分别存在于球面上对应两点$\boldsymbol{r}(t_1)$与$\boldsymbol{r}(t_2)$的球面方程（即在参数为$t_1$与$t_2$时，射线与球面存在交点）。

$$\begin{aligned} r &= ||\boldsymbol{r}(t) - \mathbf{c}|| \\ r^2 &= (\boldsymbol{r}(t) - \boldsymbol{c}) \cdot (\boldsymbol{r}(t) - \boldsymbol{c}) \\ r^2 &= (\boldsymbol{q} + t\boldsymbol{u} - \boldsymbol{c}) \cdot (\boldsymbol{q} + t\boldsymbol{u} - \boldsymbol{c}) \\ r^2 &= (\boldsymbol{q} - \boldsymbol{c} + t\boldsymbol{u}) \cdot (\boldsymbol{q} - \boldsymbol{c} + t\boldsymbol{u}) \end{aligned}$$

为了便于表达，我们设$\boldsymbol{m} = \boldsymbol{q} - \boldsymbol{c}$。

$$\begin{aligned} (\boldsymbol{m} + t\boldsymbol{u}) \cdot (\boldsymbol{m} + t\boldsymbol{u}) &= r^2 \\ \boldsymbol{m} \cdot \boldsymbol{m} + 2t\boldsymbol{m} \cdot \boldsymbol{u} + t^2\boldsymbol{u} \cdot \boldsymbol{u} &= r^2 \\ t^2\boldsymbol{u} \cdot \boldsymbol{u} + 2t\boldsymbol{m} \cdot \boldsymbol{u} + \boldsymbol{m} \cdot \boldsymbol{m} - r^2 &= 0 \end{aligned}$$

这其实就是一个一元二次方程，其中：

$$a = \boldsymbol{u} \cdot \boldsymbol{u}$$
$$b = 2(\boldsymbol{m} \cdot \boldsymbol{u})$$
$$c = \boldsymbol{m} \cdot \boldsymbol{m} - r^2$$

如果射线方向向量为单位长度，则$a = \boldsymbol{u} \cdot \boldsymbol{u} = 1$。若方程的解都含有虚部，那么射线与球体没有相交。如果有两个相同的实数解，那么射线与球体相切。若得到两个不同的实数解，则射线穿过球面，两者有两个交点。如果得到一正一负两个解，则说明射线端点位于一球体内，射线从球体内射出，负数解的交点是射线“后侧”的交点（也就是射线反向延长线与球体的交点）。这最小的正数解给出的便是与摄像机最近的相交参数。[1]

17.3.3　射线与三角形的相交检测

为了进行射线与三角形的相交检测，需要使用DirectX碰撞检测库中的TriangleTests::Intersects函数：

```
bool XM_CALLCONV
TriangleTests::Intersects(
  FXMVECTOR Origin,   // 射线的原点（端点）
  FXMVECTOR Direction, // 射线的方向向量（单位长度）
  FXMVECTOR V0, // 三角形顶点v0
  GXMVECTOR V1, // 三角形顶点v1
  HXMVECTOR V2, // 三角形顶点v2
  float& Dist ); // 射线的相交参数
```

设$\boldsymbol{r}(t) = \boldsymbol{q} + t\boldsymbol{u}$为一条射线，在满足条件$u \geqslant 0, v \geqslant 0$且$u + v \leqslant 1$时$\boldsymbol{T}(u,\ v) = \boldsymbol{v}_0 + u(\boldsymbol{v}_1 - \boldsymbol{v}_0) + v(\boldsymbol{v}_2 - \boldsymbol{v}_0)$是一个三角形（见图17.5）。我们希望求出使$\boldsymbol{r}(t) = \boldsymbol{T}(u,\ v)$（这便是射线与三角形的相交点）成立的$t$, u, v

：

$$
\begin{aligned}
&\boldsymbol{r}(t)=\boldsymbol{T}(u,\ v)\\
&\boldsymbol{q}+t\boldsymbol{u}=\boldsymbol{v}_0+u(\boldsymbol{v}_1-\boldsymbol{v}_0)+v(\boldsymbol{v}_2-\boldsymbol{v}_0)\\
&-t\boldsymbol{u}+u(\boldsymbol{v}_1-\boldsymbol{v}_0)+v(\boldsymbol{v}_2-\boldsymbol{v}_0)=\boldsymbol{q}-\boldsymbol{v}_0
\end{aligned}
$$

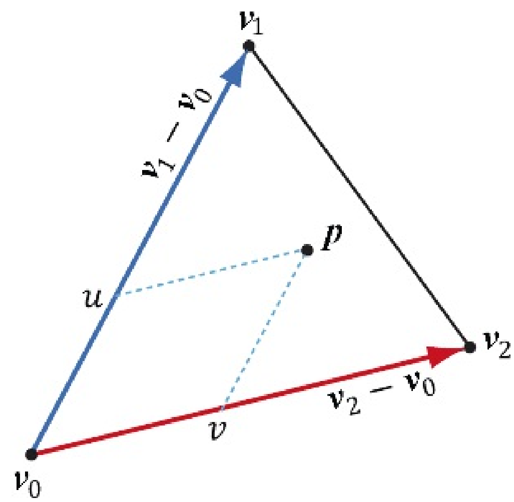


图17.5　相对于原点为$\boldsymbol{v}_0$，且坐标轴分别为$\boldsymbol{v}_1-\boldsymbol{v}_0$与$\boldsymbol{v}_2-\boldsymbol{v}_0$的非正交坐标系来说，射线与三角形相交的点$\boldsymbol{p}$在三角形所在平面上的坐标为（$u, v$）

为了便于表示，设$\boldsymbol{e}_1=\boldsymbol{v}_1-\boldsymbol{v}_0$，$\boldsymbol{e}_2=\boldsymbol{v}_2-\boldsymbol{v}_0$且$\boldsymbol{m}=\boldsymbol{q}-\boldsymbol{v}_0$，则有：

$$
-t\boldsymbol{u}+u\boldsymbol{e}_1+v\boldsymbol{e}_2=\boldsymbol{m}
$$

$$
\begin{bmatrix}\uparrow & \uparrow & \uparrow\\ -\boldsymbol{u} & \boldsymbol{e}_1 & \boldsymbol{e}_2\\ \downarrow & \downarrow & \downarrow\end{bmatrix}\begin{bmatrix}t\\ u\\ v\end{bmatrix}=\begin{bmatrix}\uparrow\\ \boldsymbol{m}\\ \downarrow\end{bmatrix}
$$

思考矩阵方程$\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$，其中，矩阵$\boldsymbol{A}$可逆。根据克莱姆法则（Cramer’s Rule）可知，$x_i=\det\boldsymbol{A}_i/\det\boldsymbol{A}$，用列向量$\boldsymbol{b}$替换矩阵$\boldsymbol{A}$中第$i$列的列向量即可求得$\boldsymbol{A}_i$。因此，

$$
t=\det\begin{bmatrix}\uparrow & \uparrow & \uparrow\\ \boldsymbol{m} & \boldsymbol{e}_1 & \boldsymbol{e}_2\\ \downarrow & \downarrow & \downarrow\end{bmatrix}/\det\begin{bmatrix}\uparrow & \uparrow & \uparrow\\ -\boldsymbol{u} & \boldsymbol{e}_1 & \boldsymbol{e}_2\\ \downarrow & \downarrow & \downarrow\end{bmatrix}
$$

$$u = \det \begin{bmatrix} \uparrow & \uparrow & \uparrow \\ -\boldsymbol{u} & \boldsymbol{m} & \boldsymbol{e}_2 \\ \downarrow & \downarrow & \downarrow \end{bmatrix} / \det \begin{bmatrix} \uparrow & \uparrow & \uparrow \\ -\boldsymbol{u} & \boldsymbol{e}_1 & \boldsymbol{e}_2 \\ \downarrow & \downarrow & \downarrow \end{bmatrix}$$

$$v = \det \begin{bmatrix} \uparrow & \uparrow & \uparrow \\ -\boldsymbol{u} & \boldsymbol{e}_1 & \boldsymbol{m} \\ \downarrow & \downarrow & \downarrow \end{bmatrix} / \det \begin{bmatrix} \uparrow & \uparrow & \uparrow \\ -\boldsymbol{u} & \boldsymbol{e}_1 & \boldsymbol{e}_2 \\ \downarrow & \downarrow & \downarrow \end{bmatrix}$$

根据等式 $\det \begin{bmatrix} \uparrow & \uparrow & \uparrow \\ \boldsymbol{a} & \boldsymbol{b} & \boldsymbol{c} \\ \downarrow & \downarrow & \downarrow \end{bmatrix} = \boldsymbol{a} \cdot (\boldsymbol{b} \times \boldsymbol{c})$ ，我们便可以将公式改写为：

$$\begin{aligned} t &= -\boldsymbol{m} \cdot (\boldsymbol{e}_1 \times \boldsymbol{e}_2) / \boldsymbol{u} \cdot (\boldsymbol{e}_1 \times \boldsymbol{e}_2) \\ u &= \boldsymbol{u} \cdot (\boldsymbol{m} \times \boldsymbol{e}_2) / \boldsymbol{u} \cdot (\boldsymbol{e}_1 \times \boldsymbol{e}_2) \\ v &= \boldsymbol{u} \cdot (\boldsymbol{e}_1 \times \boldsymbol{m}) / \boldsymbol{u} \cdot (\boldsymbol{e}_1 \times \boldsymbol{e}_2) \end{aligned}$$

为了优化运算过程，我们根据行列式的性质将矩阵中的列进行互换，在此过程中行列式的符号会发生改变（行列式的基本性质之一）：

$$\begin{aligned} t &= \boldsymbol{e}_2 \cdot (\boldsymbol{m} \times \boldsymbol{e}_1) / \boldsymbol{e}_1 \cdot (\boldsymbol{u} \times \boldsymbol{e}_2) \\ u &= \boldsymbol{m} \cdot (\boldsymbol{u} \times \boldsymbol{e}_2) / \boldsymbol{e}_1 \cdot (\boldsymbol{u} \times \boldsymbol{e}_2) \\ v &= \boldsymbol{u} \cdot (\boldsymbol{m} \times \boldsymbol{e}_1) / \boldsymbol{e}_1 \cdot (\boldsymbol{u} \times \boldsymbol{e}_2) \end{aligned}$$

此时，我们就可以计算出叉积$\boldsymbol{m} \times \boldsymbol{e}_1$与$\boldsymbol{u} \times \boldsymbol{e}_2$，并加以复用。

17.4 应用例程

本章的演示程序中将渲染出一辆轿车的网格，并允许用户通过点击鼠标右键拾取其中的三角形，而被选中的三角形则会以“突出”的高亮材质来进行绘制（见图17.6）。要以突出材质来渲染三角形，我们就需要为它准备相应的渲染项。与本书之前在初始化阶段完整定义渲染项不同，此时的渲染项信息只能在初始化时期填写一部分。这是因为我们无法知晓哪一个三角形会被拾取，因此也就更别提起始索引的位置与世界矩阵了。再者，用户也不太可能总是选中同一个三角形。所以，我们还要为渲染项的结构体添加一个Visible属性，从而令一个不可见的渲染项不被绘制出来。以下代码是PickingApp::Pick方法的部分实现细节，演示的内容是我们怎样基于用户所选的三角形来填写初始化时未设置的其余渲染项属性：

```
// 在PickingApp 类中，缓存指向被拾取三角形渲染项的指针
RenderItem* mPickedRitem;

if(TriangleTests::Intersects(rayOrigin, rayDir, v0, v1, v2, t))
{
  if(t < tmin)
  {
    // 这是当前被拾取的离摄像机最近的三角形
    tmin = t;
    UINT pickedTriangle = i;

    // 为拾取的三角形设置渲染项，这样一来我们就能用特殊的“突出”材质对它进行渲染
    mPickedRitem->Visible = true;
    mPickedRitem->IndexCount = 3;
    mPickedRitem->BaseVertexLocation = 0;

    // 被拾取的渲染项需要与被拾取的物体使用相同的世界矩阵
    mPickedRitem->World = ri->World;
    mPickedRitem->NumFramesDirty = gNumFrameResources;
```

```
    // 偏移到被拾取的三角形在网格索引缓冲区中的索引处
    mPickedRitem->StartIndexLocation = 3 * pickedTriangle;
  }
}
```

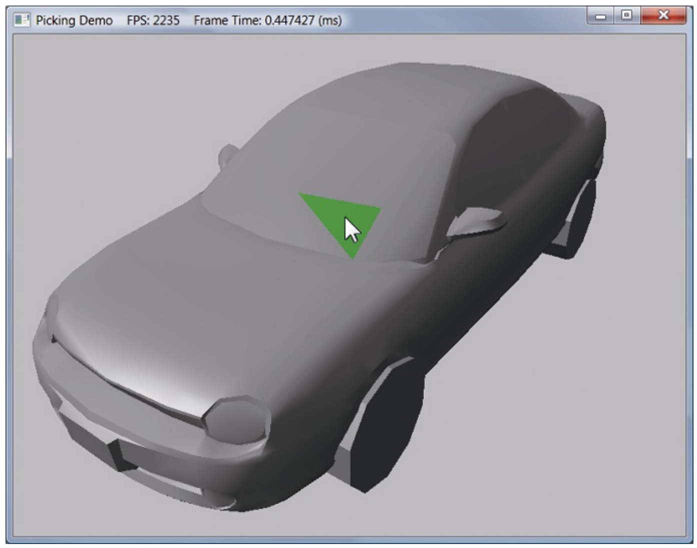


图17.6　用高亮黄色突出显示被用户拾取的三角形

在绘制完所有不透明渲染项之后，我们才会对用户选中的渲染项进行绘制。该渲染项使用的是一个颜色特殊的PSO（渲染流水线状态对象），利用透明混合技术和比较函数为D3D12_COMPARISON_FUNC_LESS_EQUAL的深度测试实现。用户拾取的三角形需要绘制两次，第二次以突出的特殊高亮材质进行渲染。如果使用的比较函数仅为D3D12_COMPARISON_FUNC_LESS，那么第二次绘制三角形时深度测试将会失败。

```
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Opaque])
;
mCommandList->SetPipelineState(mPSOs["highlight"].Get());
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Highligh
t]);
```

17.5 小结

1．拾取是一种根据用户鼠标在屏幕中选择的2D投影物体来确定其对应的3D物体的技术。

2．以观察空间的原点作为端点发出一条射线，使它经过投影窗口中与用户选择的屏幕点所对应的点，这便是拾取射线。

3．要对射线$\boldsymbol{r}(t)=\boldsymbol{q}+t\boldsymbol{u}$进行变换，可以通过使用变换矩阵来转换它的端点$\boldsymbol{q}$与方向$\boldsymbol{u}$来实现。注意，射线端点的变换结果为一点$\boldsymbol{p}$（$w=1$），而其方向的变换结果为一个向量（$w=0$）。

4．为了检验射线与物体是否相交，我们要对物体上的所有三角形一一进行射线与三角形的相交检测。如果射线与其中的一个三角形相交，则该射线也必与三角形所属的网格相交。否则，此射线与该网格并没有交集。一般来讲，我们所需的是与摄像机距离最近的三角形，这是因为如果有一些三角形重叠在射线所经路径当中，那么可能会有多个三角形与射线相交，而用户能看到的为离摄像机最近的三角形。

5．一种针对射线与网格相交检测的性能优化方法是：首先执行射线与近似于网格的包围体的相交检测。如果射线与包围体没有交集，则射线也必然不会与此三角形网格相交，因而也就不需要再进行后续的计算工作了。若射线与包围体相交，那么我们应进一步执行射线与网格的相交检测。假如射线与场景中的大多数包围体没有交集，则这个方案将为我们减少许多不必要的射线与三角形的相交检测。

17.6　练习

1．修改“Picking”（拾取）演示程序，用网格的包围球取代其中的AABB。

2．研究射线与AABB相交检测的算法。

3．假如场景中有数以千计的物体，我们还必须为实现拾取技术而执行上千次的射线与包围体的检测。那么，现请研究八叉树（octrees）这种数据结构，并解释如何利用它们来减少射线与包围体相交检测的次数。同理，该策略也可以推广到视锥体剔除技术，用来减少这个过程中视锥体与包围体相交检测的次数。

---

[1]　仔细说来，射线与球体共存在5种位置关系（当然，由于拾取射线的定义，所以将忽略其中的几种情况），另外两种分别为：两个正实数解，在射线的正方上与球体有两个交点；两个负实数解，在射线的反向延长线上与球体有两个交点。因此，说明射线的原点（端点）相对于球体的位置也很重要。读者可找寻一些带图示的教程，一目了然。

第18章　立方体贴图

本章将围绕立方体图（cube map）展开讨论，即以特殊的方式来运用这种由6个纹理所构成的基本数组。有了这项贴图技术，我们就能方便地映射天空纹理或模拟反射。

学习目标：

1．学习立方体贴图的概念并用HLSL代码对它们进行采样。

2．摸索如何利用DirectX纹理工具来创建立方体图。

3．探究如何用立方体图来模拟反射。

4．理解怎样通过立方体图渲染球体，并以此技术来模拟天空以及远山。

18.1　什么是立方体贴图

立方体贴图（cube mapping，也有译作立方体纹理映射等）的主要思路是：存储6个纹理，将它们分别看作立方体的6个面——因此而得名“立方体图”。另外，此立方体的中心点位于某坐标系的原点，且该立方体对齐于该坐标系的主轴。由于立方体纹理是轴对齐的，也就是说，它的每个面各对应于坐标系某个方向的主轴，因此我们可以根据与面相交的坐标轴方向（$\pm X$，$\pm Y$，$\pm Z$）来引用立方体图的特定面。

在Direct3D中，立方体图被表示为一个由6个元素所构成的纹理数组，即：

1．索引0援引的是与$+X$轴相交的面。

2．索引1援引的是与$-X$轴相交的面。

3．索引2援引的是与$+Y$轴相交的面。

4．索引3援引的是与$-Y$轴相交的面。

5．索引4援引的是与$+Z$轴相交的面。

6．索引5援引的是与$-Z$轴相交的面。

寻找立方体图中纹素的方法与普通的2D纹理并不相同，此时不再用2D纹理坐标来指定纹素，而是要使用3D纹理坐标：它定义了一个起

点位于原点的查找（lookup）向量$\boldsymbol{v}$。向量$\boldsymbol{v}$与立方体图相交处的纹素（见图18.1）即为$\boldsymbol{v}$的3D坐标所对应的纹素。我们在第9章中所讨论的纹理过滤思想便贯彻在向量$\boldsymbol{v}$与纹素样本间求取交点的过程当中。[1]

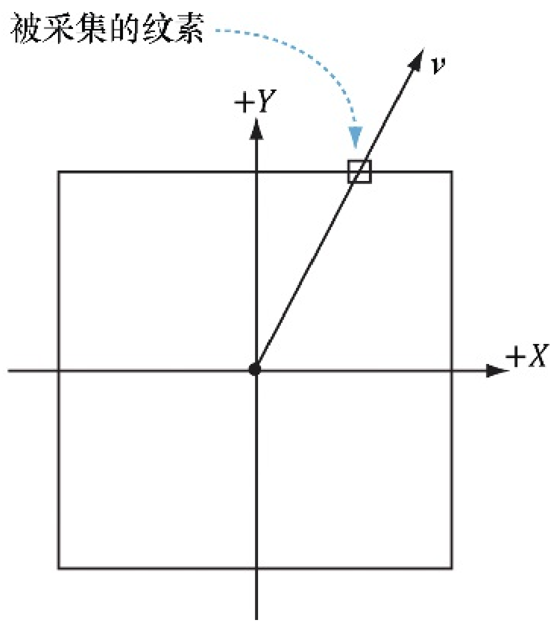


图18.1　这里为简单起见而采用2D示意图，因此图中的正方形即为3D空间中的一个立方体。图中的正方形表示一个中心位于原点、且轴对齐于某坐标系主轴的立方体图。从原点发射的向量$\boldsymbol{v}$与立方体图相交处的纹素即为采集的目标纹素。在此示意图中，与向量$\boldsymbol{v}$相交的是+Y轴上的立方体面

注意 Note

查找向量的模并不重要，关键在于它的方向。而方向相同但大小不一的两个向量在立方体图中采集的实际是同一点。

在HLSL中，立方体纹理用TextureCube类型来表示。下列代码片段详尽地展示了对立方体图进行采样的方法：

```
TextureCube gCubeMap;
SamplerState gsamLinearWrap    : register(s2);

...

// 在像素着色器中
float3 v = float3(x,y,z); // 某查找向量
float4 color = gCubeMap.Sample(gsamLinearWrap,v);
```

注意 Note

查找向量与立方体图应该位于同一空间之中。例如，若立方体图相对于世界空间而设（即立方体各面皆与世界空间的坐标轴对齐），则查找向量也应当使用世界空间坐标。

18.2　环境贴图

立方体图的主要应用是环境贴图（environment mapping，也有译作环境映射等）。其思路是：使视场角为$90°$（垂直视场角与水平视场角皆是如此）的摄像机位于场景中某物体的中心点O处。这样一来，此摄像机就能沿x、y、z三轴的正、负共6个方向进行观察，并以这6个视角来截取场景中的图像（此物体O除外）。由于视场角为$90°$，所以这6张以物体O视角截取的图像，涵盖了包围着它的整个环境。接着，我们把这6张周围环境图像存于一个立方体图中，这也正是“环境图”（environment map）这个名字的由来。换句话说，环境图就是每个面都存有周围环境图像的立方体图。[2]

根据以上描述可知，我们需要为每一个采用环境贴图的物体都创建一个环境图。虽然这种做法的效果更为精准，但也要为纹理耗费更多的内存。一种折中的实现方案是，仅在场景中的关键处截取少量环境图，为每个物体采集场景中离它们最近的环境图。在实际工作中，这种简化的做法在处理曲面物体时通常会很有效，因为体现在它们表面上的不精确反射很难被用户注意到。另一种常见的简化手段是在采集环境图时忽略场景中的某些环境图。例如，图18.2中的环境图仅采集了天空与远山这种极远处的“背景”信息，而忽略场景中的物体。尽管采集的只是背景而非整个场景的环境图，但用这种方案来创建镜面反射的时候还是很实用的。为了采集局部物体，我们要通过Direct3D来渲染环境图中的6幅图像，在18.5节中有相关的探讨。在本章的演示程序（见图18.3）中，

场景内的所有物体都采用的是如图18.2所示的那张环境图。



图18.2　图中演示的环境图是一个“拆开”后的立方体图。现在假设将这6个面重新叠回一个3D立方体，而且我们正置身于它的中心点处。这时，我们在此向四周观察就能看到周围的环境了

如果摄像机在构建环境图时所用的坐标轴方向向量为世界空间的坐标轴向量，那么就称此环境图是相对于世界空间而生成的。当然，我们可以从一个不同“原点”（例如物体的局部空间）来捕捉环境图。但是一定要保证查找向量的坐标与立方体图处于同一坐标系中。

由于立方体图仅存储纹理数据，美工就可以预先制作出纹理的内容（就像我们之前所用的2D纹理一样）。正因如此，我们也就无须通过实时渲染来计算立方体图中的图像了。即我们可以在3D场景编辑器中创建一个场景，并在编辑器中预先渲染出立方体图6个面上的图像。对于户外环境图来讲，Terragen程序（免费供个人使用）是一种比较普遍的选择，它能够创造出具有照片级真实感的室外场景来。我们为本书创建的环境图，例如图18.2所示的效果，就是借助Terragen实现的。

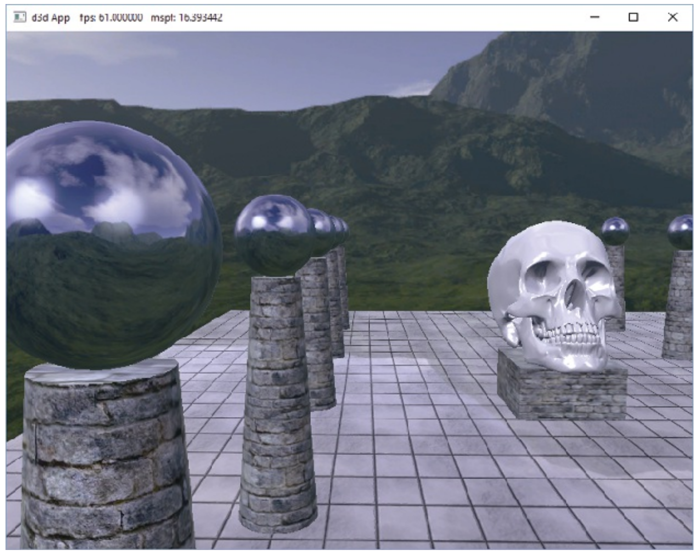


图18.3 “Cube Map”（立方体图）演示程序的运行效果

注意 Note

如果尝试使用Terragen，我们需要在Camera Settings（摄像机设置）对话框中将zoom（缩放）因子设为1.0才能使用视场角为90°的摄像机，而且还需确认图像输出的宽度与高度是相同的，以令垂直视场角与水平视场角都为90°。[3]

注意 Note

网络上有一个不错Terragen脚本（《Skybox (2D) with Terragen》），它会在当前的摄像机位置以90°的视场角渲染出6幅周围

环境的图像。

一旦利用工具创建出立方体图所用的6幅图像，我们就可以构建出含有这6幅图像的立方体图纹理了。我们所用的DDS纹理图像格式亦支持立方体图，利用texassemble工具便可以通过6幅图像构建出此格式的立方体图。下面的例子展示了如何用texassemble来创建一个立方体图（截取自texassemble的相关文档）：

```
texassemble cube -w 256 -h 256 -o cubemap.dds lobbyxpos.jpg lobbyxneg.jpg
        lobbyypos.jpg lobbyyneg.jpg lobbyzpos.jpg lobbyzneg.jpg
```

注意 Note

NVIDIA为Photoshop程序提供了用于存储DDS格式图像与立方体图的插件。

通过**Direct3D**加载并使用立方体图

正如前文所述，Direct3D通过存有6个元素的纹理数组来表示立方体图。演示程序中的DDS纹理加载代码（DDSTextureLoader.h/.cpp）已经支持对立方体图的加载，而且载入其他类型的纹理也不在话下。加载

代码会检测出含有一幅立方体图的DDS文件，并创建纹理数组将每个立方体面的纹理数据载入相应的元素中。

```
auto skyTex = std::make_unique<Texture>();
skyTex->Name = "skyTex";
skyTex->Filename = L"Textures/grasscube1024.dds";
ThrowIfFailed(DirectX::CreateDDSTextureFromFile12(md3dDevice.Get(),
    mCommandList.Get(), skyTex->Filename.c_str(),
    skyTex->Resource, skyTex->UploadHeap));
```

在为立方体图纹理资源创建SRV（着色器资源视图）时，应将其维度指定为D3D12_SRV_DIMENSION_TEXTURECUBE且使用TextureCube属性：

```
D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING
;
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURECUBE;
srvDesc.TextureCube.MostDetailedMip = 0;
srvDesc.TextureCube.MipLevels = skyTex->GetDesc().MipLevels;
srvDesc.TextureCube.ResourceMinLODClamp = 0.0f;
srvDesc.Format = skyTex->GetDesc().Format;
md3dDevice->CreateShaderResourceView(skyTex.Get(), &srvDesc, hDescriptor);
```

18.3　绘制天空纹理

我们能够利用环境图绘制天空纹理。首先，要围绕整个场景来创建一个巨大的球体。为了营造出遥不可及的远山以及天穹的错觉，我们以图18.4所展示的方法，通过环境图来为球体绘制纹理。这种方法的思路就是将纹理图投影到球面之上。

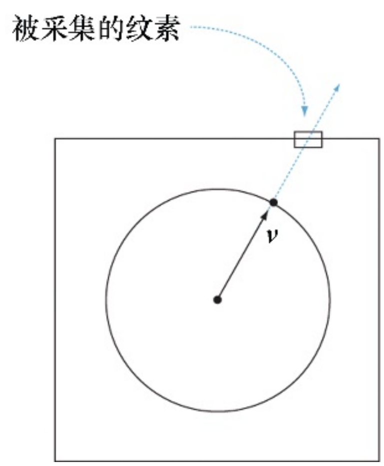


图18.4　为了简单起见，这里展示的是2D示意图；因此，图中的正方形即为3D空间中的立方体（图），而圆形即为3D空间中的（天空）球体。假设天空球与环境图都以同一个空间中的原点为中心。接下来，为了向球面上的点投影纹理，我们把端点为原点的向量作为查找向量射向球面。以此将立方体图投影到球面上

假设天空球（sky sphere）距摄像机是无限远的（即它的中心位于世界空间的原点，但半径无穷大），这样一来，无论摄像机移到场景中的哪个角落，我们都无法更加接近或愈发远离天空球面。要实现这种无限远的天穹，可以在世界空间里简单地将天空球的中心置于摄像机上，使它总是以摄像机为中心。由于天空球会随着摄像机而移动，所以摄像机并不会更接近于球面。若非如此，在将摄像机移近天空表面的时候，

整个戏法必然会被拆穿，因为我们这里所用的模拟天空的把戏很容易被用户识破。

实现天穹效果的着色器文件如下：

```
//*********************************************************************
// Sky.hlsl的作者为Frank Luna (C) 2015 版权所有
//*********************************************************************

// 包含公用的HLSL代码
#include "Common.hlsl"

struct VertexIn
{
  float3 PosL  : POSITION;
  float3 NormalL : NORMAL;
  float2 TexC  : TEXCOORD;
};

struct VertexOut
{
  float4 PosH : SV_POSITION;
  float3 PosL : POSITION;
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout;

  // 用局部顶点的位置作为立方体图的查找向量
  vout.PosL = vin.PosL;

  // 把顶点变换到世界空间
  float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);

  // 总是以摄像机作为天空球的中心
  posW.xyz += gEyePosW;

  // 设置z = w，从而使z/w = 1（即令球面总是位于远平面）
  vout.PosH = mul(posW, gViewProj).xyww;

  return vout;
}

float4 PS(VertexOut pin) : SV_Target
```

```
{
  return gCubeMap.Sample(gsamLinearWrap, pin.PosL);
}
```

渲染天空的着色器程序与绘制普通物体的着色器程序（Default.hlsl）有着明显的区别。但是它们所用的根签名是相同的，因此也就不必在绘制的过程中改变根签名了。由于Default.hlsl和Sky.hlsl所共用的代码部分已经移入Common.hlsl文件，所以它们的代码并不存在交集。为便于读者参考，这里给出Common.hlsl文件的代码：

```
//*****************************************************************************
// Common.hlsl的作者为Frank Luna (C) 2015版权所有
//*****************************************************************************

// 光源数量的默认值
#ifndef NUM_DIR_LIGHTS
  #define NUM_DIR_LIGHTS 3
#endif

#ifndef NUM_POINT_LIGHTS
  #define NUM_POINT_LIGHTS 0
#endif

#ifndef NUM_SPOT_LIGHTS
  #define NUM_SPOT_LIGHTS 0
#endif

// 包含光照所需的结构体与函数
#include "LightingUtil.hlsl"
// 每种材质所用到的各种常量数据
struct MaterialData
{
  float4  DiffuseAlbedo;
  float3  FresnelR0;
  float  Roughness;
  float4x4 MatTransform;
  uint   DiffuseMapIndex;
  uint   MatPad0;
  uint   MatPad1;
  uint   MatPad2;
```

```
};

TextureCube gCubeMap : register(t0);

// 仅着色器模型5.1+才支持的纹理数组。与Texture2DArray不同的是，此数组可以由大小不一、格式
// 各异的纹理组成。因此，这使它比普通的纹理数组也更为灵活
Texture2D gDiffuseMap[4] : register(t1);

// 将此结构化缓冲区置于space1中，以使纹理数组不会与这些资源相重叠。而纹理数组则会占用寄
// 存器t0, t1, …, t3中的space0
StructuredBuffer<MaterialData> gMaterialData : register(t0, space1);

SamplerState gsamPointWrap    : register(s0);
SamplerState gsamPointClamp    : register(s1);
SamplerState gsamLinearWrap    : register(s2);
SamplerState gsamLinearClamp   : register(s3);
SamplerState gsamAnisotropicWrap : register(s4);
SamplerState gsamAnisotropicClamp : register(s5);

// 每一帧中所用到的各种常量数据
cbuffer cbPerObject : register(b0)
{
  float4x4 gWorld;
  float4x4 gTexTransform;
  uint gMaterialIndex;
  uint gObjPad0;
  uint gObjPad1;
  uint gObjPad2;
};

// 绘制过程中所用到的杂项常量数据
cbuffer cbPass : register(b1)
{
  float4x4 gView;
  float4x4 gInvView;
  float4x4 gProj;
  float4x4 gInvProj;
  float4x4 gViewProj;
  float4x4 gInvViewProj;
  float3 gEyePosW;
  float cbPerObjectPad1;
  float2 gRenderTargetSize;
  float2 gInvRenderTargetSize;
  float gNearZ;
  float gFarZ;
```

```
  float gTotalTime;
  float gDeltaTime;
  float4 gAmbientLight;

  // 对于每个以MaxLights 为光源数量最大值的对象而言，索引[0, NUM_DIR_LIGHTS]表
示的是
  // 方向光光源，索引[NUM_DIR_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHTS)表示的
是点光源，索引[NUM_DIR_
  // LIGHTS+NUM_POINT_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHT+NUM_SPOT_LIGH
TS)表
  // 示的是聚光灯光源
  Light gLights[MaxLights];
};
```

注意 Note

早期的应用程序总是要先绘制天空，再用它作为替代品去清理（填写）渲染目标以及深度/模板缓冲区。但是，“ATI Radeon HD 2000 Programming Guide”（ATI Radeon HD 2000编程指南）现在并不建议用户按此方式进行处理：首先，要使内部硬件得到更深层次的优化而表现得更为出色，需要显式地清理深度/模板缓冲区。这种使用情景与渲染目标比较相似。其次，天空的大部分区域会被建筑物或地形这样的其他几何体遮挡住。因此，若率先绘制天空，则将会把许多资源浪费在无效像素的绘制上——这些像素将被后续所绘制的、离摄像机更近的物体所遮蔽。总之，建议读者对缓冲区进行手动清理，并把绘制天空的工作放在最后。

绘制天空需要使用与众不同的着色器程序，继而也就会用到新的

PSO（流水线状态对象）。因此，我们在绘制代码中把天空作为独立的层进行渲染：

```
// 绘制不透明的渲染项
mCommandList->SetPipelineState(mPSOs["opaque"].Get());
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)
    RenderLayer::Opaque]);

// 绘制天空渲染项
mCommandList->SetPipelineState(mPSOs["sky"].Get());
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)
    RenderLayer::Sky]);
```

除此之外，渲染天空还需要采用一些不同的渲染状态。由于摄像机位于天空球内，在禁用背面剔除（若将逆时针绕序的三角形定为正面朝向，则可不禁用此功能）之余，我们还要将深度比较函数改为LESS_EQUAL，以令天空球能够顺利地通过深度测试：

```
D3D12_GRAPHICS_PIPELINE_STATE_DESC skyPsoDesc = opaquePsoDesc;

// 摄像机位于天空球内，所以要关闭剔除功能
skyPsoDesc.RasterizerState.CullMode = D3D12_CULL_MODE_NONE;

// 确认深度测试函数为LESS_EQUAL而非仅为LESS。否则的话，如果深度缓冲区中的数据都被清理为1，
// 则归一化深度值为z = 1 (NDC，用规格化设备坐标所表示)的深度项将在深度测试中失败
skyPsoDesc.DepthStencilState.DepthFunc = D3D12_COMPARISON_FUNC_LESS_EQUAL;
skyPsoDesc.pRootSignature = mRootSignature.Get();
skyPsoDesc.VS =
{
        reinterpret_cast<BYTE*>(mShaders["skyVS"]->GetBufferPointer()),
        mShaders["skyVS"]->GetBufferSize()
};
skyPsoDesc.PS =
{
        reinterpret_cast<BYTE*>(mShaders["skyPS"]->GetBufferPointer()),
        mShaders["skyPS"]->GetBufferSize()
};
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(
  &skyPsoDesc, IID_PPV_ARGS(&mPSOs["sky"])));
```

18.4　模拟反射

在第8章中，我们学习了镜面高光的实现原理：光源发出的光照射在物体表面（界面）上，并基于菲涅耳效应与表面的粗糙度反射到观察者的眼中。但是，照射到物体表面（界面）上的光并非仅从光源传来的直射光，而是由于散射与反弹的原因从各个方向照射而来“混合光”。事实上，我们也已将环境光项加入光照方程之中，从而模拟间接漫反射光照。在本节中，我们将展示如何运用环境图去模拟来自周围环境的镜面反射（specular reflection）。通过镜面反射，我们就能够观察到基于菲涅耳效应而从物体表面（界面）反射来的光。这里还有一个相关的高级主题，但在本书中不会加以讨论，这也是一种通过立方体图计算来自周围环境漫反射光的方法（读者可以参考《GPU Gems 2》中的Chapter 10. Real-Time Computation of Dynamic Irradiance Environment Map）。

当我们为构建环境图而关于点O渲染场景时，实则是在点O处记录来自四面八方的光照数据。换句话说，环境图存储的是从各个方向照射到点O处的光照值，因此我们可以把环境图上的每个纹素都看作一个光源。通过这些数据便可以近似地计算出来自周围环境光的镜面反射情况。为加深理解，可参阅图18.5。来自入射方向$\boldsymbol{I}$的环境光，根据菲涅耳效应经界面反射，以方向$\boldsymbol{v}=\boldsymbol{E}-\boldsymbol{p}$进入观察者的眼中。通过查找向量$\boldsymbol{r}=\mathrm{reflect}(-\boldsymbol{v},\boldsymbol{n})$对环境立方体图进行采样以获取环境光。这一系列设定使得界面具有类似于镜面的属性，即观察者查看点$\boldsymbol{p}$便可看到点$\boldsymbol{p}$处反射的周围环境。

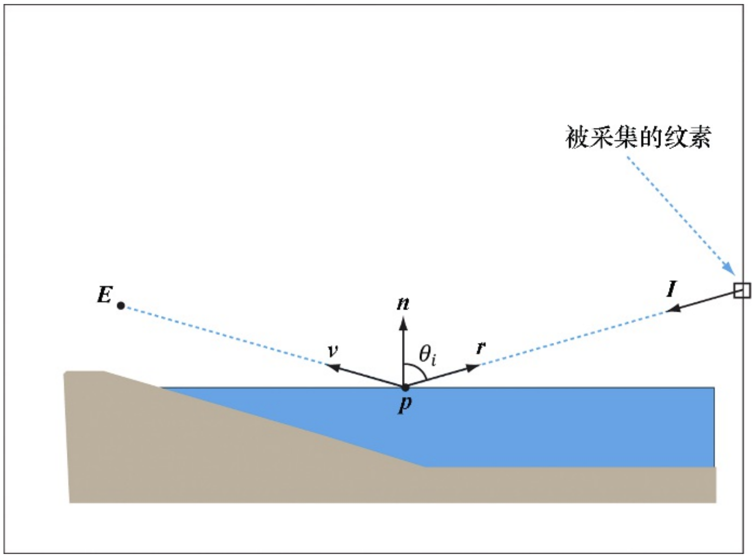


图18.5　图中的点$\boldsymbol{E}$是观察点，$\boldsymbol{n}$是点$\boldsymbol{P}$处的表面法线。通过查找向量$\boldsymbol{r}$对立方体图进行采样，即可获得存有从$\boldsymbol{P}$以方向$\boldsymbol{v}$反射入观察者眼中光线数据的纹素

计算每个像素的反射向量并用它来对环境图进行采样：

```
const float shininess = 1.0f - roughness;

// 加入镜面反射数据
float3 r = reflect(-toEyeW, pin.NormalW);
float4 reflectionColor = gCubeMap.Sample(gsamLinearWrap, r);
float3 fresnelFactor = SchlickFresnel(fresnelR0, pin.NormalW, r);
litColor.rgb += shininess * fresnelFactor * reflectionColor.rgb;
```

由于讨论的是与反射相关的内容，所以自然少不了提及菲涅耳效应。它基于表面的材质属性、光向量（反射向量）与法线之间的夹角，从而确定从环境中反射到观察者眼中的光量。除此之外，我们还要根据材质的光泽度增减反射值，即粗糙材质反射的光量较低，即便如此，这些较小的反射值仍不可忽略。

从图18.6中可以看出，对于平整的表面来讲，通过环境贴图实现的反射效果并不是很好。

这是由于反射向量不能给出明确的位置关系，因为它不含有具体的位置信息，而我们需要的恰好就是反射光线及其与环境图的交点。光线具有位置与方向这两种属性，而向量仅能提供此两者中的方向信息。从图中我们可以看到，按道理来讲，$\boldsymbol{q}(t)=\boldsymbol{p}+t\boldsymbol{r}\boldsymbol{q}'(t)=\boldsymbol{p}'+t\boldsymbol{r}$两束反射光线，它们分别交于立方体图的两个不同纹素，因而反射出的光线颜色应当会有差异。然而，由于两束光线采用的是相同的方向向量$\boldsymbol{r}$，而又仅用方向向量$\boldsymbol{r}$去实现立方体图的纹素查找工作，因此，若分别在$\boldsymbol{E}$与$\boldsymbol{E}'$两点处观察，则会看到映射到$\boldsymbol{p}$与$\boldsymbol{p}'$处的纹素是相同的。对于平滑的物体来讲，这种环境贴图的瑕疵会更加明显。而相对于曲面物体而言，该环境贴图的短板将更不易察觉，因为曲面的弯曲度会导致其反射向量方向各异。

对此，一种解决方案是给环境图关联一个代理几何体（proxy geometry）。例如，假设有一用于立方体空间的环境图。此时，我们就能给该环境图关联一个与其空间大小近似的轴对齐包围盒。图18.7所示的是，我们是如何令一条射线与包围盒相交，以求取比反射向量$\boldsymbol{r}$指向的纹素更为精准的查找向量$\boldsymbol{v}$。如果将与立方体图相关联的包围盒输入到着色器中（例如通过常量缓冲区），就可以在像素着色器中进行射线与包围盒的相交检测，据此就能以改良的查找向量在像素着色器中对立方体图进行采样。[4]

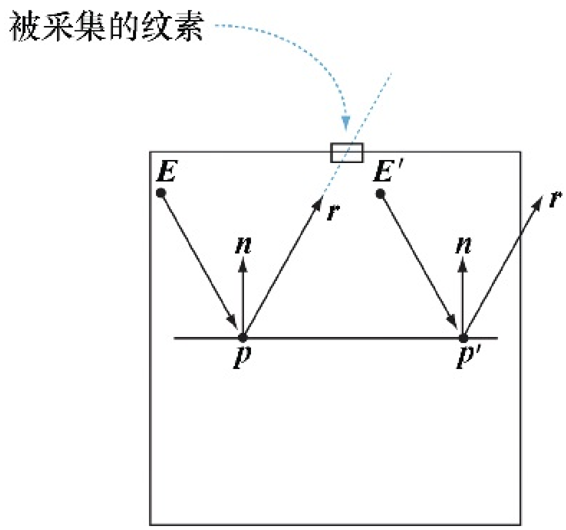


图18.6　当观察位置分别位于$\boldsymbol{E}$与$\boldsymbol{E}'$时，对应于平面上$\boldsymbol{p}$与$\boldsymbol{p}'$两个不同点的反射向量

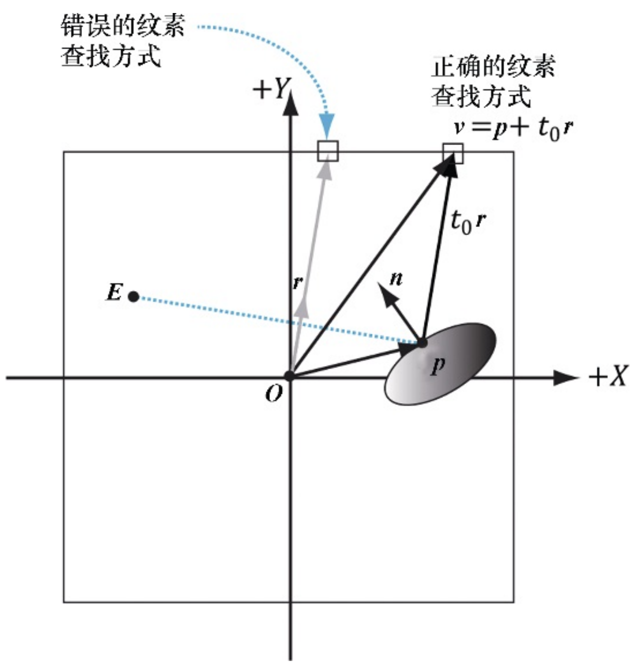


图18.7　在这种情况下，我们不再用反射向量$\boldsymbol{r}$来作为立方体图的查找向量，而是用射线与包围盒的交点$\boldsymbol{v}=\boldsymbol{p}+t_0\boldsymbol{r}$来代替。注意，由于点$\boldsymbol{p}$与包围盒代理几何体的中心位置有关，所以上述交点可作为此立方体图的查找向量

以下函数所展示的是立方体图查找向量的计算方法：

```
float3 BoxCubeMapLookup(float3 rayOrigin, float3 unitRayDir,
```

```
            float3 boxCenter, float3 boxExtents)
{
  // 本实现基于《Real-Time Rendering（实时渲染）》第3版中16.7.1节所描述的slab method[5]

  // 令射线的端点与包围盒的中心位置有关
  float3 p = rayOrigin - boxCenter;

  // AABB（轴对齐包围盒）中第i个slab射线与平面相交检测的公式为
  //
  // t1 = (-dot(n_i, p) + h_i)/dot(n_i, d) = (-p_i + h_i)/d_i
  // t2 = (-dot(n_i, p) - h_i)/dot(n_i, d) = (-p_i - h_i)/d_i

  // 将所有的slab 都进行向量化处理，并按射线与平面的相交检测公式进行计算
  float3 t1 = (-p+boxExtents)/unitRayDir;
  float3 t2 = (-p-boxExtents)/unitRayDir;

  // 寻找每个坐标轴上的最大值。由于我们假设射线就位于包围盒内，因而只希望求取最大的相交参数，即t值
  float3 tmax = max(t1, t2);

  // 求取tmax所有分量中的最小值
  float t = min(min(tmax.x, tmax.y), tmax.z);

  // 由于点p是相对于包围盒的中心位置，所以可将它用于计算立方体图的查找向量
  return p + t*unitRayDir;
}
```

18.5　动态立方体图

到目前为止，我们所描述的都是静态立方体图，它所存储的都是预先绘制好的固定图像。这种工作方式对于某些情景来说是比较合理的，而且开销较小。但是，如果我们希望在场景中创建一些会移动的动态角色，那么这种方案就不太合适了。若采用事先生成的立方体图，我们就不能用它来捕捉那些动态物体，这也就意味着不能绘制出动态物体的反射镜像。为了克服这种限制，就应在运行时动态地构建立方体图。即我们在每一帧都要将摄像机置于场景之内，以它作为立方体图的原点，沿着坐标轴共6个方向将场景分六次逐个渲染到立方体图的对应面上（见图18.8）。由于在每一帧都会重建立方体图，因此能捕捉到场景中的动态物体及其动态的反射镜像（见图18.9）。

注意 Note

动态地渲染立方体图开销会比较大，因为每一帧都需要将场景绘制到6个渲染目标之中。因此，我们要试着将场景中需要用到动态立方体图的地方降到最少。比如，我们可以只为突出场景中的关键物品时才使用动态反射。而为动态反射镜像要求不高的次要物品采用静态立方体图。一般来讲，动态立方体贴图常用的是256×256像素这样低分辨率的立方体图，这样做可以减少要处理的像素数量（fill rate，像素填充率）。

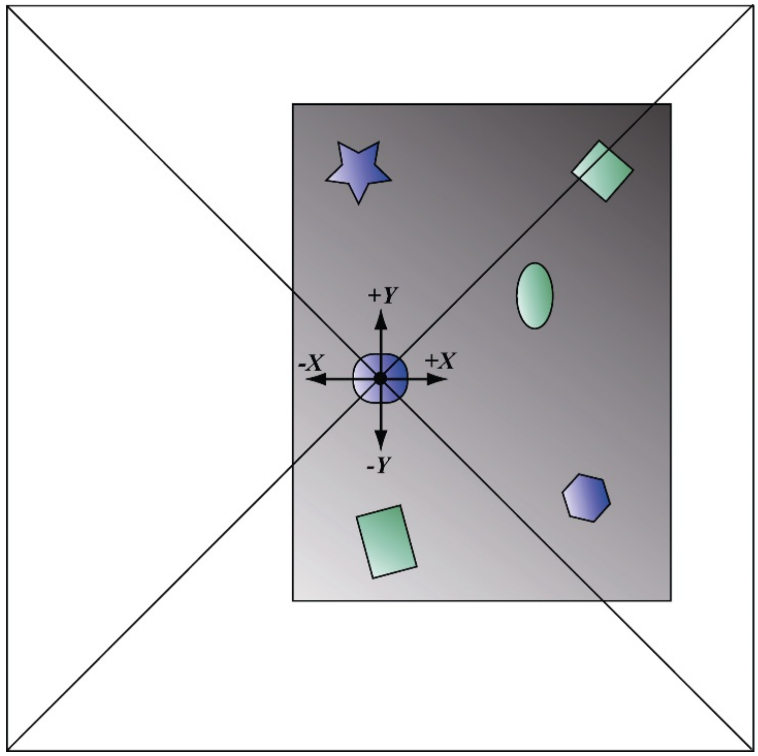


图18.8　摄像机在场景中的位置O，正处于所希望生成动态立方体图的物体的中心处。以视场角为90°的摄像机沿坐标轴的6个方向将场景分别渲染一次，以此来截取整个周围环境的图像

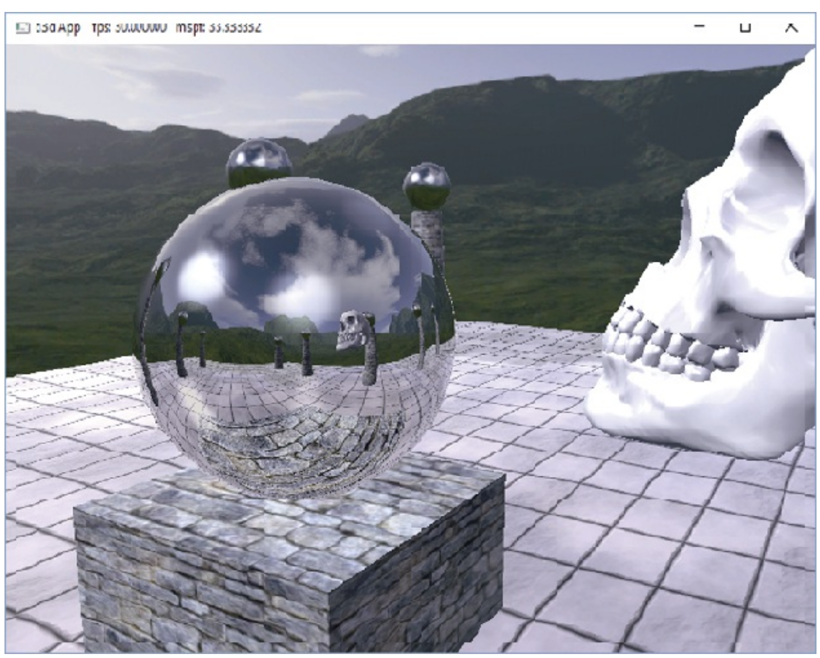


图18.9 “Dynamic CubeMap”（动态立方体图）演示程序所呈现的动态反射效果。骷髅头绕场景中间的球体旋转，它的镜像也动态地反映在该球面上。既然我们是自行绘制立方体图，因此也就可以将局部物体的镜像渲染在球面上，如立柱、柱上的球体以及地面

18.5.1 动态立方体图辅助类

为了便于动态地渲染立方体图，我们创建了以下CubeRenderTarget类。此类内部封装了立方体图的实际ID3D12Resource对象、与该资源所对应的各种描述符，以及用于渲染立方体图的的其他有关数据。

```
class CubeRenderTarget
{
public:
  CubeRenderTarget(ID3D12Device* device,
    UINT width, UINT height,
    DXGI_FORMAT format);

  CubeRenderTarget(const CubeRenderTarget& rhs)=delete;
  CubeRenderTarget& operator=(const CubeRenderTarget& rhs)=delete;
  ~CubeRenderTarget()=default;

  ID3D12Resource* Resource();
  CD3DX12_GPU_DESCRIPTOR_HANDLE Srv();
  CD3DX12_CPU_DESCRIPTOR_HANDLE Rtv(int faceIndex);

  D3D12_VIEWPORT Viewport()const;
  D3D12_RECT ScissorRect()const;

  void BuildDescriptors(
    CD3DX12_CPU_DESCRIPTOR_HANDLE hCpuSrv,
    CD3DX12_GPU_DESCRIPTOR_HANDLE hGpuSrv,
    CD3DX12_CPU_DESCRIPTOR_HANDLE hCpuRtv[6]);

  void OnResize(UINT newWidth, UINT newHeight);

private:
  void BuildDescriptors();
  void BuildResource();
```

```
private:

  ID3D12Device* md3dDevice = nullptr;

  D3D12_VIEWPORT mViewport;
  D3D12_RECT mScissorRect;

  UINT mWidth = 0;
  UINT mHeight = 0;
  DXGI_FORMAT mFormat = DXGI_FORMAT_R8G8B8A8_UNORM;

  CD3DX12_CPU_DESCRIPTOR_HANDLE mhCpuSrv;
  CD3DX12_GPU_DESCRIPTOR_HANDLE mhGpuSrv;
  CD3DX12_CPU_DESCRIPTOR_HANDLE mhCpuRtv[6];

  Microsoft::WRL::ComPtr<ID3D12Resource> mCubeMap = nullptr;
};
```

18.5.2 构建立方体图资源

通过创建具有6个元素的纹理数组（每个元素都对应着一个立方体图的面），便可以构建出立方体图纹理。欲渲染立方体图则必须设置D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET标志。我们通过以下方法来构建立方体图资源。

```
void CubeRenderTarget::BuildResource()
{
  D3D12_RESOURCE_DESC texDesc;
  ZeroMemory(&texDesc, sizeof(D3D12_RESOURCE_DESC));
  texDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
  texDesc.Alignment = 0;
  texDesc.Width = mWidth;
  texDesc.Height = mHeight;
  texDesc.DepthOrArraySize = 6;
  texDesc.MipLevels = 1;
  texDesc.Format = mFormat;
  texDesc.SampleDesc.Count = 1;
  texDesc.SampleDesc.Quality = 0;
  texDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
```

```
  texDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET;

  ThrowIfFailed(md3dDevice->CreateCommittedResource(
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
    D3D12_HEAP_FLAG_NONE,
    &texDesc,
    D3D12_RESOURCE_STATE_GENERIC_READ,
    nullptr,
    IID_PPV_ARGS(&mCubeMap)));
}
```

18.5.3 分配额外的描述符堆空间

为了渲染立方体图，还需要新添加6个渲染目标视图，使之与立方体图的各个面一一对应。另外，还要附加一个深度/模板缓冲区，因此我们必须重写（override）`D3DApp::CreateRtvAndDsvDescriptorHeaps`方法来为这些额外的描述符分配描述符堆。

```
void DynamicCubeMapApp::CreateRtvAndDsvDescriptorHeaps()
{
  // 为立方体渲染目标添加6个RTV（渲染目标视图）
  D3D12_DESCRIPTOR_HEAP_DESC rtvHeapDesc;
  rtvHeapDesc.NumDescriptors = SwapChainBufferCount + 6;
  rtvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
  rtvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
  rtvHeapDesc.NodeMask = 0;
  ThrowIfFailed(md3dDevice->CreateDescriptorHeap(
    &rtvHeapDesc, IID_PPV_ARGS(mRtvHeap.GetAddressOf())));

  // 为立方体渲染目标新增1个DSV（深度/模板视图）
  D3D12_DESCRIPTOR_HEAP_DESC dsvHeapDesc;
  dsvHeapDesc.NumDescriptors = 2;
  dsvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_DSV;
  dsvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
  dsvHeapDesc.NodeMask = 0;
  ThrowIfFailed(md3dDevice->CreateDescriptorHeap(
    &dsvHeapDesc, IID_PPV_ARGS(mDsvHeap.GetAddressOf())));
```

```
  mCubeDSV = CD3DX12_CPU_DESCRIPTOR_HANDLE(
    mDsvHeap->GetCPUDescriptorHandleForHeapStart(),
    1,
    mDsvDescriptorSize);
}
```

除此之外，还需新增一个SRV（着色器资源视图），以便在生成立方体图之后将它绑定为着色器的输入数据。

描述符的句柄都要传入CubeRenderTarget::BuildDescriptors方法，它会保存一份句柄的副本并为它们创建相应的视图。

```
auto srvCpuStart = mSrvDescriptorHeap->GetCPUDescriptorHandleForHeapStart(
);
auto srvGpuStart = mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart(
);
auto rtvCpuStart = mRtvHeap->GetCPUDescriptorHandleForHeapStart();

// 位列交换链的描述符之后的立方体图RTV
int rtvOffset = SwapChainBufferCount;

CD3DX12_CPU_DESCRIPTOR_HANDLE cubeRtvHandles[6];
for(int i = 0; i < 6; ++i)
  cubeRtvHandles[i] = CD3DX12_CPU_DESCRIPTOR_HANDLE(
    rtvCpuStart, rtvOffset + i, mRtvDescriptorSize);

mDynamicCubeMap->BuildDescriptors(
  CD3DX12_CPU_DESCRIPTOR_HANDLE(
    srvCpuStart, mDynamicTexHeapIndex, mCbvSrvDescriptorSize),
CD3DX12_GPU_DESCRIPTOR_HANDLE(
  srvGpuStart, mDynamicTexHeapIndex, mCbvSrvDescriptorSize),
cubeRtvHandles);

void CubeRenderTarget::BuildDescriptors(CD3DX12_CPU_DESCRIPTOR_HANDLE hCpu
Srv,
                        CD3DX12_GPU_DESCRIPTOR_HANDLE hGpuSrv,
                        CD3DX12_CPU_DESCRIPTOR_HANDLE hCpuRtv[6])
{
  // 保存对描述符的引用
  mhCpuSrv = hCpuSrv;
  mhGpuSrv = hGpuSrv;
```

```
  for(int i = 0; i < 6; ++i)
    mhCpuRtv[i] = hCpuRtv[i];

  // 创建描述符
  BuildDescriptors();
}
```

18.5.4　构建描述符

在上一节中，我们为描述符分配了相应的描述符堆空间，并缓存了对描述符的引用。但是到目前为止，仍没有为任何资源真正地创建描述符。因此，现在的当务之急是为立方体图资源创建一个SRV，以便能够在像素着色器中对它进行采样。另外，还需要为立方体图纹理数组中的每个元素创建一个渲染目标视图，借此对立方体图的每个面依次进行绘制。创建所需视图的方法如下。

```
void CubeRenderTarget::BuildDescriptors()
{
  D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
  srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPI
NG;
  srvDesc.Format = mFormat;
  srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURECUBE;
  srvDesc.TextureCube.MostDetailedMip = 0;
  srvDesc.TextureCube.MipLevels = 1;
  srvDesc.TextureCube.ResourceMinLODClamp = 0.0f;

  // 为整个立方体图资源创建SRV
  md3dDevice->CreateShaderResourceView(mCubeMap.Get(), &srvDesc,
    mhCpuSrv);

  // 为每个立方体面创建RTV
  for(int i = 0; i < 6; ++i)
  {
    D3D12_RENDER_TARGET_VIEW_DESC rtvDesc;
    rtvDesc.ViewDimension = D3D12_RTV_DIMENSION_TEXTURE2DARRAY;
    rtvDesc.Format = mFormat;
    rtvDesc.Texture2DArray.MipSlice = 0;
```

```
    rtvDesc.Texture2DArray.PlaneSlice = 0;

    // 表示现要为第i个元素创建渲染目标视图
    rtvDesc.Texture2DArray.FirstArraySlice = i;

    // 仅为数组中的每一个元素创建一个视图
    rtvDesc.Texture2DArray.ArraySize = 1;

    // 为立方体图的第i个面创建RTV
    md3dDevice->CreateRenderTargetView(mCubeMap.Get(), &rtvDesc,
    mhCpuRtv[i]);
  }
}
```

18.5.5 构建深度缓冲区

一般来讲，立方体图的各面与主后台缓冲区的分辨率是不同的。因此，要渲染立方体图的诸面，我们就需要采用与立方体图面分辨率大小相匹配的深度缓冲区。考虑到每次只渲染立方体的一个面，所以立方体图的渲染工作仅需1个深度缓冲区即可。我们可以通过下列代码来构建新添的深度缓冲区及其DSV。

```
void DynamicCubeMapApp::BuildCubeDepthStencil()
{
  // 创建深度/模板缓冲区及其视图
  D3D12_RESOURCE_DESC depthStencilDesc;
  depthStencilDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
  depthStencilDesc.Alignment = 0;
  depthStencilDesc.Width = CubeMapSize;
  depthStencilDesc.Height = CubeMapSize;
  depthStencilDesc.DepthOrArraySize = 1;
  depthStencilDesc.MipLevels = 1;
  depthStencilDesc.Format = mDepthStencilFormat;
  depthStencilDesc.SampleDesc.Count = 1;
  depthStencilDesc.SampleDesc.Quality = 0;
  depthStencilDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
  depthStencilDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL;

  D3D12_CLEAR_VALUE optClear;
```

```
  optClear.Format = mDepthStencilFormat;
  optClear.DepthStencil.Depth = 1.0f;
  optClear.DepthStencil.Stencil = 0;
  ThrowIfFailed(md3dDevice->CreateCommittedResource(
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
    D3D12_HEAP_FLAG_NONE,
    &depthStencilDesc,
    D3D12_RESOURCE_STATE_COMMON,
    &optClear,
    IID_PPV_ARGS(mCubeDepthStencilBuffer.GetAddressOf())));

  // 以资源自身的格式为整个资源的mip 0层级创建描述符
  md3dDevice->CreateDepthStencilView(
    mCubeDepthStencilBuffer.Get(), nullptr, mCubeDSV);

  // 将资源从初始状态转换为深度缓冲区
  mCommandList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER::Transition(
    mCubeDepthStencilBuffer.Get(),
    D3D12_RESOURCE_STATE_COMMON,
    D3D12_RESOURCE_STATE_DEPTH_WRITE));
}
```

18.5.6　立方体图的视口与裁剪矩形

由于立方体图各面与主后台缓冲区的分辨率不一致，因此需要定义一个新的视口以及裁剪矩形来“对准拍摄”立方体图面。

```
CubeRenderTarget::CubeRenderTarget(ID3D12Device* device,
                  UINT width, UINT height,
                  DXGI_FORMAT format)
{
  md3dDevice = device;

  mWidth = width;
  mHeight = height;
  mFormat = format;

  mViewport = { 0.0f, 0.0f, (float)width, (float)height, 0.0f, 1.0f };
  mScissorRect = { 0, 0, width, height };

  BuildResource();
```

```
}

D3D12_VIEWPORT CubeRenderTarget::Viewport()const
{
    return mViewport;
}

D3D12_RECT CubeRenderTarget::ScissorRect()const
{
    return mScissorRect;
}
```

18.5.7 设置立方体图摄像机

前文曾讲到，生成立方体图的方法是，把视场角为$90°$（垂直方向和水平方向的视场角都是$90°$）的摄像机架设在场景中某物体O的中心点。再使摄像机分别对准x、y、z三轴的正、负共6个方向，并以这6种视角来摄取场景图片（除了物体O）。为了便于此流程的实现，我们以给定的位置(x, y, z)为中心生成了6台摄像机，它们分别负责立方体图中一个面上的图像截取工作。

```
Camera mCubeMapCamera[6];
void DynamicCubeMapApp::BuildCubeFaceCamera(float x, float y, float z)
{
  // 生成指定位置处的立方体图
  XMFLOAT3 center(x, y, z);
  XMFLOAT3 worldUp(0.0f, 1.0f, 0.0f);

  // 沿着每一个坐标轴方向进行观察
  XMFLOAT3 targets[6] =
  {
    XMFLOAT3(x + 1.0f, y, z), // +X
    XMFLOAT3(x - 1.0f, y, z), // -X
    XMFLOAT3(x, y + 1.0f, z), // +Y
    XMFLOAT3(x, y - 1.0f, z), // -Y
    XMFLOAT3(x, y, z + 1.0f), // +Z
    XMFLOAT3(x, y, z - 1.0f)  // -Z
  };
```

```
  // 除了+Y/-Y，其他方向上的上向量均用世界空间中的上向量(0,1,0)表示。在+Y/-Y这两个方向上，我
  // 们分别要沿着+Y或-Y进行观察，因此便需要用一个与众不同的"上"向量
  XMFLOAT3 ups[6] =
  {
    XMFLOAT3(0.0f, 1.0f, 0.0f),   // +X
    XMFLOAT3(0.0f, 1.0f, 0.0f),   // -X
    XMFLOAT3(0.0f, 0.0f, -1.0f),  // +Y
    XMFLOAT3(0.0f, 0.0f, +1.0f),  // -Y
    XMFLOAT3(0.0f, 1.0f, 0.0f),   // +Z
    XMFLOAT3(0.0f, 1.0f, 0.0f)    // -Z
  };

  for(int i = 0; i < 6; ++i)
  {
    mCubeMapCamera[i].LookAt(center, targets[i], ups[i]);
    mCubeMapCamera[i].SetLens(0.5f*XM_PI, 1.0f, 0.1f, 1000.0f);
    mCubeMapCamera[i].UpdateViewMatrix();
  }
}
```

由于渲染立方体的不同面要动用不同的摄像机，因此每个立方体面都需要拥有一组它自己独有的PassConstants（渲染过程常量）。好在这项工作并不复杂，只要在创建帧资源时将PassConstants的个数再增加6个即可。

```
void DynamicCubeMapApp::BuildFrameResources()
{
  for(int i = 0; i < gNumFrameResources; ++i)
  {
    mFrameResources.push_back(std::make_unique<FrameResource>(md3dDevice.Get(),
      7, (UINT)mAllRitems.size(), (UINT)mMaterials.size()));
  }
}
```

元素0对应于主渲染过程，而元素1～6则与立方体诸面相对应。

为每一个立方体图面设置常量数据的方法如下：

```
void DynamicCubeMapApp::UpdateCubeMapFacePassCBs()
{
  for(int i = 0; i < 6; ++i)
  {
    PassConstants cubeFacePassCB = mMainPassCB;

    XMMATRIX view = mCubeMapCamera[i].GetView();
    XMMATRIX proj = mCubeMapCamera[i].GetProj();

    XMMATRIX viewProj = XMMatrixMultiply(view, proj);
    XMMATRIX invView = XMMatrixInverse(&XMMatrixDeterminant(view),
    view);
    XMMATRIX invProj = XMMatrixInverse(&XMMatrixDeterminant(proj),
    proj);
    XMMATRIX invViewProj = XMMatrixInverse(&XMMatrixDeterminant(viewProj),
 viewProj);

    XMStoreFloat4x4(&cubeFacePassCB.View, XMMatrixTranspose(view));
    XMStoreFloat4x4(&cubeFacePassCB.InvView,
    XMMatrixTranspose(invView));
    XMStoreFloat4x4(&cubeFacePassCB.Proj, XMMatrixTranspose(proj));
    XMStoreFloat4x4(&cubeFacePassCB.InvProj,
    XMMatrixTranspose(invProj));
    XMStoreFloat4x4(&cubeFacePassCB.ViewProj,
    XMMatrixTranspose(viewProj));
    XMStoreFloat4x4(&cubeFacePassCB.InvViewProj, XMMatrixTranspose(invView
Proj));
    cubeFacePassCB.EyePosW = mCubeMapCamera[i].GetPosition3f();
    cubeFacePassCB.RenderTargetSize =
      XMFLOAT2((float)CubeMapSize, (float)CubeMapSize);
    cubeFacePassCB.InvRenderTargetSize =
      XMFLOAT2(1.0f / CubeMapSize, 1.0f / CubeMapSize);

    auto currPassCB = mCurrFrameResource->PassCB.get();

    // 元素1~6中存储的是立方体图（6个面）渲染过程中所用的常量缓冲区
    currPassCB->CopyData(1 + i, cubeFacePassCB);
  }
}
```

18.5.8　对立方体图进行绘制

针对此演示程序，我们设定了3个渲染层：

```
enum class RenderLayer : int
{
  Opaque = 0,
  OpaqueDynamicReflectors,
  Sky,
  Count
};
```

OpaqueDynamicReflectors渲染层含有图18.9所示的通过动态立方体图技术来反射局部动态物体的中心球体。首先将场景绘制到立方体图的每个面上，但是不包括中心球体自身。这也就意味着只需把不透明物体层以及天空层渲染到立方体图即可。

```
void DynamicCubeMapApp::DrawSceneToCubeMap()
{
  mCommandList->RSSetViewports(1, &mDynamicCubeMap->Viewport());
  mCommandList->RSSetScissorRects(1, &mDynamicCubeMap->ScissorRect());

  // 将立方体图资源转换为RENDER_TARGET（渲染目标）
  mCommandList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER::Transition(
    mDynamicCubeMap->Resource(),
    D3D12_RESOURCE_STATE_GENERIC_READ,
    D3D12_RESOURCE_STATE_RENDER_TARGET));

   UINT passCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(PassCo
nstants));

  // 针对立方体图的每个面……
  for(int i = 0; i < 6; ++i)
  {
    // 清理后台缓冲区以及深度缓冲区
    mCommandList->ClearRenderTargetView(
      mDynamicCubeMap->Rtv(i), Colors::LightSteelBlue, 0, nullptr);
    mCommandList->ClearDepthStencilView(mCubeDSV,
      D3D12_CLEAR_FLAG_DEPTH | D3D12_CLEAR_FLAG_STENCIL,
      1.0f, 0, 0, nullptr);

    // 指定将要渲染的缓冲区
    mCommandList->OMSetRenderTargets(1, &mDynamicCubeMap->Rtv(i),
      true, &mCubeDSV);
```

```
    // 为当前的立方体图面绑定对应的渲染过程常量缓冲区，这样一来，我们就可以使用正确的视图矩阵以
    // 及投影矩阵来绘制此立方体图
    auto passCB = mCurrFrameResource->PassCB->Resource();
    D3D12_GPU_VIRTUAL_ADDRESS passCBAddress =
      passCB->GetGPUVirtualAddress() + (1+i)*passCBByteSize;
    mCommandList->SetGraphicsRootConstantBufferView(1, passCBAddress);

    DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)
    RenderLayer::Opaque]);

    mCommandList->SetPipelineState(mPSOs["sky"].Get());
    DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)
    RenderLayer::Sky]);

    mCommandList->SetPipelineState(mPSOs["opaque"].Get());
  }

  // 将立方体图资源转换回GENERIC_READ状态，以便在着色器中读取纹理数据
  mCommandList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER::Transition(
    mDynamicCubeMap->Resource(),
    D3D12_RESOURCE_STATE_RENDER_TARGET,
    D3D12_RESOURCE_STATE_GENERIC_READ));
}
```

向立方体图渲染场景之后，我们还要像以前那样设置主渲染目标并绘制场景，但是别忘了还要将动态立方体图绘制到中心球体上。

```
...
DrawSceneToCubeMap();

// 设置主渲染目标

mCommandList->RSSetViewports(1, &mScreenViewport);
mCommandList->RSSetScissorRects(1, &mScissorRect);

// 根据资源的用处而转换其状态
mCommandList->ResourceBarrier(1,
  &CD3DX12_RESOURCE_BARRIER::Transition(
  CurrentBackBuffer(),
  D3D12_RESOURCE_STATE_PRESENT,
  D3D12_RESOURCE_STATE_RENDER_TARGET));

// 清理后台缓冲区与深度缓冲区
```

```
mCommandList->ClearRenderTargetView(CurrentBackBufferView(),
  Colors::LightSteelBlue, 0, nullptr);
mCommandList->ClearDepthStencilView(
  DepthStencilView(),
  D3D12_CLEAR_FLAG_DEPTH | D3D12_CLEAR_FLAG_STENCIL,
  1.0f, 0, 0, nullptr);

// 指定将要渲染的缓冲区
mCommandList->OMSetRenderTargets(1,
  &CurrentBackBufferView(), true, &DepthStencilView());

auto passCB = mCurrFrameResource->PassCB->Resource();
mCommandList->SetGraphicsRootConstantBufferView(1,
  passCB->GetGPUVirtualAddress());

// 为动态反射层OpaqueDynamicReflectors使用动态立方体图
CD3DX12_GPU_DESCRIPTOR_HANDLE dynamicTexDescriptor(
  mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());
dynamicTexDescriptor.Offset(mSkyTexHeapIndex + 1,
    mCbvSrvDescriptorSize);
mCommandList->SetGraphicsRootDescriptorTable(3, dynamicTexDescriptor);

DrawRenderItems(mCommandList.Get(),
mRitemLayer[(int)RenderLayer::OpaqueDynamicReflectors]);

// 为其他物体（包括天空）使用静态“背景”立方体图
mCommandList->SetGraphicsRootDescriptorTable(3, skyTexDescriptor);

DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)
    RenderLayer::Opaque]);

mCommandList->SetPipelineState(mPSOs["sky"].Get());
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)
    RenderLayer::Sky]);

// 根据资源的用途而转换其状态
mCommandList->ResourceBarrier(1,
  &CD3DX12_RESOURCE_BARRIER::Transition(
  CurrentBackBuffer(),
  D3D12_RESOURCE_STATE_RENDER_TARGET,
  D3D12_RESOURCE_STATE_PRESENT));
...
```

18.6　用几何着色器绘制动态立方体图

在上一节中，我们（以不同的视角）将场景反复绘制6次，并依次渲染到每个立方体图的面上，以此来生成立方体图。绘制调用是有开销的，所以应尽量减少调用次数。Direct3D 10有一名为“CubeMapGS”的示例，它通过几何着色器仅需绘制一遍场景即可渲染好一幅立方体图。在本节中，我们的主题便是讲解这一示例的工作方式。注意，尽管这里演示的是Direct3D 10版本的相关代码，但是其中所用的策略在Direct3D 12中依然适用，而且代码移植起来也比较容易。

首先，该例程为整个纹理数组（而不是按每个面都单独分开的纹理）创建了一个渲染目标视图。

```
// 创建6个面的整体渲染目标视图
D3D10_RENDER_TARGET_VIEW_DESC DescRT;
DescRT.Format = dstex.Format;
DescRT.ViewDimension = D3D10_RTV_DIMENSION_TEXTURE2DARRAY;
DescRT.Texture2DArray.FirstArraySlice = 0;
DescRT.Texture2DArray.ArraySize = 6;
DescRT.Texture2DArray.MipSlice = 0;
V_RETURN( pd3dDevice->CreateRenderTargetView(
g_pEnvMap, &DescRT, &g_pEnvMapRTV ) );
```

要运用这种绘制方式，还需要一个由6个深度缓冲区构成的“立方体图”（即每个深度缓冲区都对应于一个面）。为整个深度缓冲区纹理数组创建深度/模板视图的过程如下。

```
// 为整个立方体创建深度/模板视图
D3D10_DEPTH_STENCIL_VIEW_DESC DescDS;
DescDS.Format = DXGI_FORMAT_D32_FLOAT;
DescDS.ViewDimension = D3D10_DSV_DIMENSION_TEXTURE2DARRAY;
```

```
DescDS.Texture2DArray.FirstArraySlice = 0;
DescDS.Texture2DArray.ArraySize = 6;
DescDS.Texture2DArray.MipSlice = 0;
V_RETURN( pd3dDevice->CreateDepthStencilView(
g_pEnvMapDepth, &DescDS, &g_pEnvMapDSV ) );
```

接下来将上述渲染目标视图与深度/模板视图绑定到渲染流水线的OS（输出合并）阶段。

```
ID3D10RenderTargetView* aRTViews[ 1 ] = { g_pEnvMapRTV };
pd3dDevice->OMSetRenderTargets(sizeof(aRTViews)/sizeof(aRTViews[0]),
aRTViews, g_pEnvMapDSV );
```

完成这些工作后，我们就已经将渲染目标数组的视图以及深度/模板缓冲区数组的视图绑定至OM阶段，随后就将对每个数组切片（array slice）同时进行渲染。

至此，场景已被渲染1次，且6个观察矩阵所构成的数组（每一个矩阵都用于按立方体图面的对应方向进行观察）已在常量缓冲区中就位。几何着色器会将输入的三角形复制6次，并依次赋予一个渲染目标数组切片。通过设置系统值SV_RenderTargetArrayIndex，我们就能把三角形赋予到渲染目标数组切片之中。此系统值是仅用于设置几何着色器输出的整数索引值，它指定了图元应当被绘制到的渲染目标数组切片的索引。而且，只有当渲染目标视图为数组资源时，才可用此系统值。

```
struct PS_CUBEMAP_IN
{
  float4 Pos : SV_POSITION;   // 投影坐标
  float2 Tex : TEXCOORD0;     // 纹理坐标
  uint RTIndex : SV_RenderTargetArrayIndex;
};

[maxvertexcount(18)]
void GS_CubeMap( triangle GS_CUBEMAP_IN input[3],
```

```
inout TriangleStream CubeMapStream )
{
  // 针对每个三角形…
  for( int f = 0; f < 6; ++f )
  {
    // 计算屏幕坐标
    PS_CUBEMAP_IN output;

    // 将第f个三角形赋予第f个渲染目标
    output.RTIndex = f;

    // 针对三角形的每个顶点…
    for( int v = 0; v < 3; v++ )
    {
      // 把顶点变换到第f个立方体面的观察空间
      output.Pos = mul( input[v].Pos, g_mViewCM[f] );

      // 将顶点变换到齐次裁剪空间
      output.Pos = mul( output.Pos, mProj );

      output.Tex = input[v].Tex;
      CubeMapStream.Append( output );
    }
    CubeMapStream.RestartStrip();
  }
}
```

可以看出，按照上述方法，仅需渲染一遍场景就能为立方体图的每个面绘制好相应的图像，而不必再逐面进行共6次渲染。

注意 Note

我们在此仅梳理了该示例的主要思路，如要了解“CubeMapGS”的具体细节，可查阅此Direct3D 10例程的源代码。

这个例程所展示的方案很有意思，既演示了多渲染目标的同时绘制，又给出了`SV_RenderTargetArrayIndex`系统值的使用方法。然而，此方法并非完美无缺，它暴露出的两个缺陷使它失色不少。

1．它使用几何着色器来输出大量的数据。在第12章中我们曾提到过：当几何着色器输出大量的数据时，它的效率极低。因此，以输出多顶点为目的来使用几何着色器会对程序的整体性能造成破坏。

2．在一般的场景中，某个三角形是不会与第二个立方体图面相重叠的（回顾图18.8）。因此，复制三角形并将其渲染至每个立方体的面上极其浪费资源，这是因为6个面之中有5个面的三角形需要被裁剪掉。不可否认的是，我们在本章中所用的例程确实简化了将整个场景渲染至每个立方体图面上的过程。然而在真实的应用程序（非演示程序）中，我们会使用视锥体剔除技术（第16章），仅将可见的物体渲染到特定的立方体图面之上。而视锥体剔除是在物体层级中执行，所以几何体着色器并不能实现此技术。

但从另一方面来说，在渲染包围场景的网格时，这种策略的效果超群。例如，假设有一动态的天空模拟系统，基于每天的不同时段，云层会飘动，天空的颜色也会发生改变。由于天空时时在变化，我们是不能预先烘焙（prebaked，指将光照或反射等效果添加至物体的纹理之上，以此改善渲染性能）出立方体图纹理来反射天空景象的，因此不得不使用动态立方体图。由于天空网格包围着整个场景，也就是说，它在立方体图的**6**个面内都是可见的。因此，这种情况完全打破了上面所列第二条论点的束缚。此时，该几何着色器方案就能依靠将6次绘制调用降为1

次而占优。当然，还要同时保证此几何着色器的用法不会严重破坏程序的性能。

注意 Note

NVIDIA公司对Maxwell架构进行的最新优化，使几何着色器向多个渲染目标复制几何体的操作不会造成过大的性能损失。尽管写作本书之时，这些特性还没有在Direct3D 12中实现，但相信在不久的将来，Direct3D能够为此而进行更新。

18.7　小结

1．立方体图由6个纹理组成，我们把它们分别视作立方体的每一个面。在Direct3D 12中，可以通过`ID3D12Resource`接口将立方体图表示为具有6个元素的纹理数组。而在HLSL中，立方体图由`TextureCube`类型表示。我们使用3D纹理坐标来指定立方体图上的纹素，它定义了一个以立方体图中心为起点的3D查找向量$\boldsymbol{v}$。该向量与立方体图相交处的纹素即为$\boldsymbol{v}$的3D坐标所对应的纹素。

2．环境图即为在某点处（以不同视角）对周围环境截取的6张图像，而这些图像最终会存于一个立方体图之中。通过环境图我们就能方便地渲染天空或模拟反射。

3．通过texassemble工具便可以用6个单独的图像创建出立方体图，并以DDS图像的格式存于文件之中。由于立方体图存有6个耗费大量内存的2D纹理，因此DDS压缩格式是上佳之选。

4．预先烘焙的立方体图既不能截取场景中的移动对象，也无法采集在它生成时还不曾存在的物体。为了克服这种限制，我们需要在运行时动态地构建立方体图。也就是说，我们在每一帧都要将摄像机架设在场景中某处，以此作为立方体图的原点，并沿着每个坐标轴方向将场景分6次渲染至每个立方体图的面上。因为每一帧都要重新构建立方体图，所以就能截取到动态对象以及环境中的每一样物体。动态立方体图的开销极大，因此应当谨慎地将它们用于关键物品的渲染。

5．我们可以将纹理数组的渲染目标视图绑定至渲染流水线的OM阶段，也能对纹理数组中的每一个数组切片同时进行渲染。利用系统值SV_RenderTargetArrayIndex便可以把三角形赋予特定的渲染目标数组切片。假设现有一个纹理数组的渲染目标视图，我们利用SV_RenderTargetArrayIndex系统值即可一次性渲染整个场景来动态地生成立方体图，而不必再对每个面一一进行绘制（共6次）。但是，这个策略并非在任何情况下都优于利用视锥体剔除共需渲染场景6次的方法。

18.8　练习

1．在“Cube Map”（立方体图）演示程序中尝试不同的`FresnelR0`值以及材质粗糙度数据。再试着令该例程中的柱子和立方体反射周围的环境。

2．寻找从某个环境中截取的6幅图像（可以从网络中搜索立方体图的图像，或用Terragen这种程序来自行制作），并通过texassemble工具将它们合并为一幅立方体图。最后以“Cube Map”演示程序来验证所制作的立方体图。

3．电介质（dielectric）是一种能够使光线发生折射的透明材质，原理如图18.10所示。当光线照射到电介质时，一部分会被反射，另一部分则会根据斯涅尔折射定律（Snell’s Law of Refraction）发生折射。我们用折射率n_1与n_2来确定光线的偏折程度：

1．如果$n_1 = n_2$，那么$\boldsymbol{\theta}_1 = \boldsymbol{\theta}_2$（不发生折射）。

2．如果$n_2 > n_1$，那么$\boldsymbol{\theta}_2 < \boldsymbol{\theta}_1$（折射光线靠近法线）。

3．如果$n_1 > n_2$，那么$\boldsymbol{\theta}_2 > \boldsymbol{\theta}_1$（折射光线远离法线）。

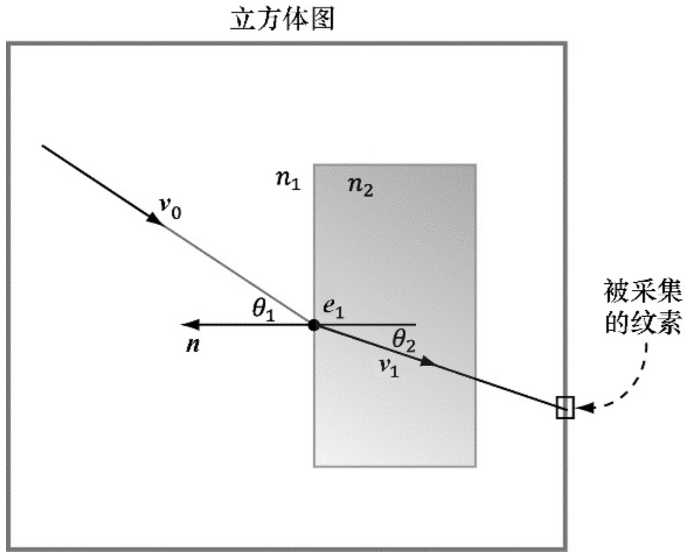


图18.10　光线沿入射向量v_0在折射率为n_1的介质中传播。待光线照射到折射率为n_2的透明材质时，将按向量v_1的方向发生折射。此时，我们把折射向量v_1作为查找立方体图纹素的向量。这个过程很像alpha混合透明处理（alpha blending transparency），只是后者不会让入射向量发生偏折

如此一来，在图18.10中，由于$n_2 > n_1$，因此光线在进入电介质块这部分区域时，折射光线向法线方向偏折。从物理学上来讲，光线在离开此区域时还会再次发生折射，但对于实时图形的绘制来说，一般只针对首次进入电介质区域时发生的光线折射进行建模。HLSL用内部函数`refract`来计算折射向量：

```
float3 refract(float3 incident, float3 normal, float eta);
```

第一个参数入射向量即入射光线向量（图18.10中的v_0），第二个参数法向量为指向电介质表面外侧的表面法线（图18.10中的n），第三个参数是折射率之比n_1/n_2。真空中的折射率为1.0，还有一些常见的折射率，如水是1.33、玻璃为1.51。在这个习题中，修改“Cube Map”演示程序，将其中模拟的反射替换为折射效果（见图18.11），在这个过程

中，我们或许还需适当的调整`MaterialData:: Roughness`数值。另外，请分别尝试设置`eta = 1.0`、`eta = 0.95`和`eta = 0.9`，并观察效果。

4．在光源经反射而产生镜面高光的过程中，粗糙度会影响镜面反射的发散程度。因此，越粗糙的表面就越能得到模糊的反射效果，因为这会使环境图中的多个样本以同样的角度均匀地散射到观察者的眼中。试研究以环境图（environment map）来模拟模糊反射（blurry reflection）的技术。

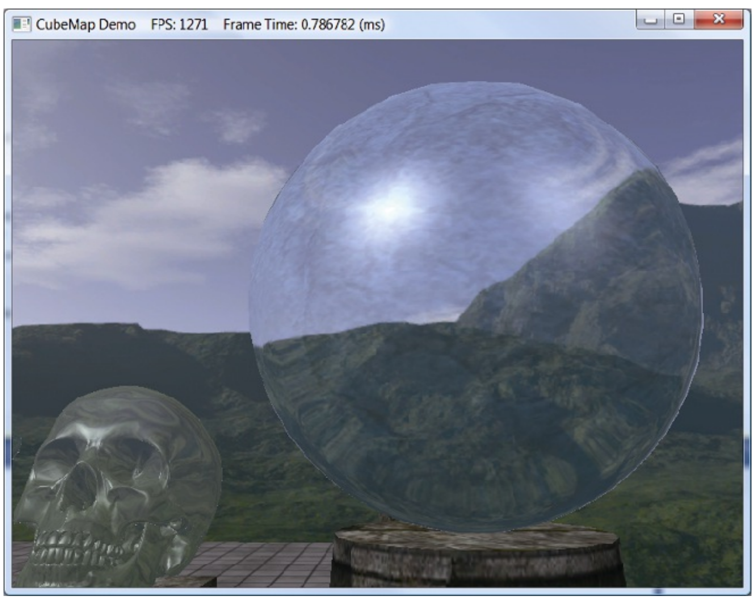


图18.11　以折射代替反射的“Cube Map”例程效果

---

[1]　向物体映射立方体图的过程叫作立方体贴图，类似的还有纹理贴图（向物体映射纹理图）等。

[2]　读者可以回顾玩360°全景图的过程。

[3]　最新版的设置方式与旧版有些区别，详见官方文档。

[4]　简而言之，环境图的映射方式是以其立方体中心为端点，向立方体发出射线采集纹素。对于立方体内的平面而言，在同一入射角观察不同点本应看到不同的图像（即采集不同的纹理图纹素），但是由于同入射角所用的loopup（查找）向量是相同的，所以在相同入射角看不同的点，看到的是相同的内容。图18.6是“理想”的采集示意图，图18.7是解决方案。如果再加一张“实际”的有误采集示意图（或者指出直接与图18.1进行对比），会更明晰。

[5]　slab即一对平行平面，一个立方体可视作3组slab。此算法的大意是将包围盒划分为3组slab，2D图看起来是井字形。再将射线与slab的相交关系转换为坐标的比较，以此来确定光线是否与包围盒相交。

第19章　法线贴图

我们曾在第9章中介绍过纹理贴图，此技术通过令图像映射在网格三角形之上，使之呈现出更加完美的细节。但是到目前为止，用于在三角形中进行插值的法向量仍然被定义在粒度较大的顶点级别。本章的重头戏就是学习一种指定更高分辨率（即细节程度更高的）曲面法线的常用方法。如果采用分辨率更高的曲面法线，便可以在网格几何体的细节保持不变的情况下，使光照的效果得到提升。

学习目标：

1．理解为什么需要法线贴图。

2．探索如何存储法线图。

3．学习如何创建法线图。

4．探究法线图中存储的法向量是相对于哪种坐标系而定义的，而这种坐标系又与3D三角形的物体空间坐标系有何关联。

5．学习如何在顶点着色器与像素着色器中实现法线贴图。

19.1 使用法线贴图的动机

图19.1是出自第18章中立方体贴图演示程序的有关场景，唯独其中锥形圆柱的镜面高光看起来似乎有些不太对劲——与砖块纹理的凹凸有致相比，它们看起来平滑得不甚自然。这是由于纹理表面下的网格几何体太过平滑，我们只是简单地把凹凸不平的砖块材质绘制于光滑的柱面而已。然而，又因光照是基于网格几何体（特别是插值顶点法线）而非纹理图像来进行计算的，所以导致光照效果并不完全与纹理保持一致。

在理想的情况下，我们应当对网格几何体进行镶嵌化处理，以令砖块按纹理表面下的几何体来进行建模，使它们像真实的砖块那样错落有致、纹路斑驳。这样一来，光照和纹理的效果就能达到统一。硬件曲面细分确实能够在这种情形中派上用场，但是我们仍需以某种方式来为镶嵌器指定生成顶点所需的法线（利用插值法线并不能提升法线的分辨率）。

另一种可行的解决方案是将光照细节直接烘焙（bake）到纹理之中。但是，在光源可移动的情况下，这种方法还是不能奏效，这是因为在光源移动的过程中，其中的纹素颜色却始终保持不变。

因此，我们的目标就变为寻找动态光照（dynamic lighting）的实现方法，以使纹理图和光照都可以同时展现出各自更佳的细节。又由于纹理的丰富细节是与生俱来的，因而问题也就自然而然地转向寻求与纹理贴图有关的解决方案之上。图19.1与图19.2各展示了同样的场景，但后

者使用了法线贴图（normal mapping）技术，根据图片我们可以明显地看出还是动态光照与砖块纹理更配。

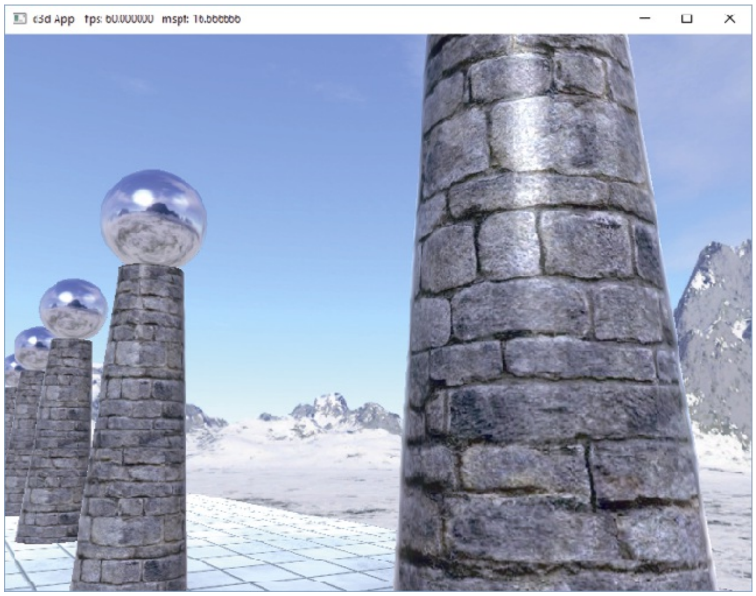


图19.1　表现出光滑平整效果的镜面高光

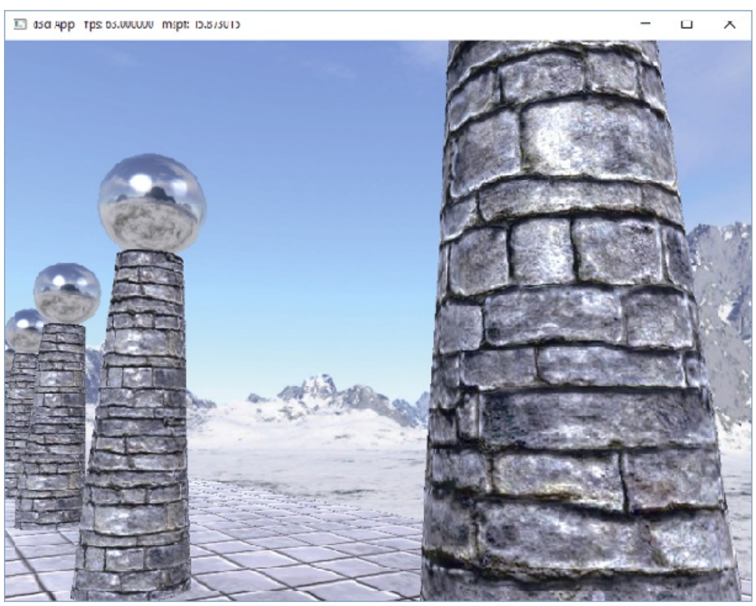


图19.2　表现出凹凸有致效果的镜面高光

19.2　什么是法线贴图

法线图（normal map）本质上也是一种纹理，但其中每一个纹素所存储的并非RGB数据，红、绿、蓝3种分量依次存储的是压缩后的x、y、z坐标，这些坐标定义的即是法向量。也就是说，法线图中的每个像素内都存储了一条法向量。图19.3所示为对一张法线图进行可视化处理后的效果。

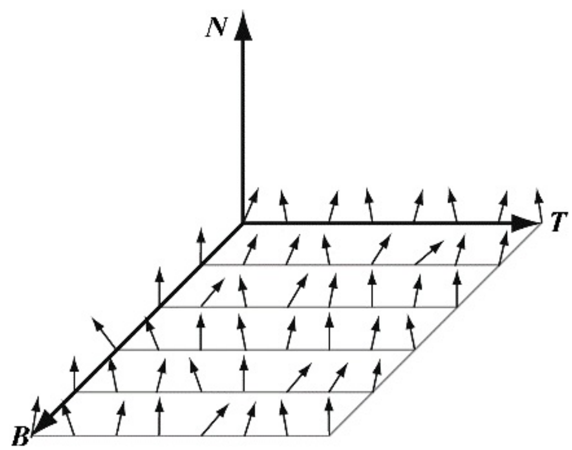


图19.3　存于法线图中的法线，它们位于由$\boldsymbol{T}$（x轴）、$\boldsymbol{B}$（y轴）、$\boldsymbol{N}$（z轴）3个向量所定义的纹理空间坐标系之中。向量$\boldsymbol{T}$指向纹理图像的右侧水平方向，向量$\boldsymbol{B}$指向纹理图像的下侧垂直方向，向量$\boldsymbol{N}$则正交于纹理平面

为了便于讲解，假设以下示例中所用的是24位图像格式，即将每个颜色分量都存于1字节之中，因此，每个颜色分量的取值范围都为[0,255]（至于32位的图像格式，其中的alpha分量可以保留不用，也可以用于存储其他的标量数据，如地形的高度图（heightmap）或高光图（specular map，也称为镜面反射图等）。当然，若使用浮点格式就不必再对坐标进行压缩了，但是这将耗费更多的内存。）

注意 Note

如图19.3所示，其中的向量大多都近似平行于z轴。换言之，这些向量的z值取得了3种坐标分量中的最大值。因此，当把法线图视作彩色图像时（也就是按每个法线的RGB值进行绘制），整体会呈现蓝色。这是因为z坐标被存于蓝色通道之中，而大部分法线又取得了较大的z值，所以蓝色会占据图像的大部分区域。

那么，又该如何将单位向量压缩为上述格式呢？首先要注意到的是“单位向量”一词，这就是说，每个坐标的范围都被限定在[−1, 1]。如果以平移及缩放的手段将其区间变换至[0, 1]，并乘以255，再截断（truncate）小数部分，则最终得到的将是[0, 255]的某个整数。即，如果x为一个在[−1, 1]区间的坐标，则$f(x)$函数得到的整数部分将处于范围[0, 255]之中。在这里，f的定义为：

$$f(x) = (0.5x + 0.5) \cdot 255$$

因此，为了在24位图像中存储单位向量，仅需将每个坐标用函数f进行变换，再把变换后的坐标写入纹理图中对应的颜色通道即可。

接下来的问题是怎样实现压缩处理的逆操作，即给出一个在范围[0, 255]内的压缩后的纹理坐标，我们以某种方法将它复原为[−1, 1]区间内的值。事实上，通过f的反函数即可方便地解决此问题，经过简单的变

换，将得到：

$$f^{-1}(x) = \frac{2x}{255} - 1$$

即，如果x是范围在[0, 255]的一个整数，则$f^{-1}(x)$得到的结果是范围[−1, 1]区间的一个浮点数。

我们不必亲自动手参与压缩处理，因为借助一款Photoshop插件就能将图片轻松转换为法线图，所以只是在像素着色器中对法线图进行采样时，还是要实现解压缩变换过程中的一些步骤。我们用类似于下列的语句在着色器中对法线图进行采样：

```
float3 normalT = gNormalMap.Sample( gTriLinearSam, pin.Tex );
```

颜色向量normalT将获取归一化分量构成的坐标(r, g, b)，其中$0 \leqslant r, g, b \leqslant 1$。

可见，坐标的解压工作现已完成一部分了（即压缩坐标已除以255，由位于[0, 255]的整数变换到[0, 1] 区间之内的浮点数）。接下来，我们就要用函数$g: [0,\ 1] \rightarrow [-1,\ 1]$通过平移与缩放来将每个分量由范围[0, 1]变换到区间[−1, 1]，以此实现整个解压缩操作。此函数的定义为：

$$g(x) = 2x - 1$$

在代码中，我们用此函数来处理每一个颜色分量：

```
// 将每个分量从[0,1]解压缩至[-1,1]。
    normalT = 2.0f*normalT - 1.0f;
```

由于标量1.0根据规则会被扩充为向量(1, 1, 1)，因此该表达式会按分量逐个进行计算。

如果用压缩纹理格式来存储法线图，那么BC7（`DXGI_FORMAT_BC7_UNORM`）图像格式的质量最佳，因为它大幅减少了因压缩法线图而导致的误差。对于BC6与BC7格式而言，DirectX SDK给出了名为“BC6HBC7EncoderDecoder11”的相应示例。我们可通过这个程序将纹理文件转换为BC6或BC7格式。

19.3　纹理空间/切线空间

现在我们来考虑将纹理映射到3D三角形上的过程。为了便于讨论，假设在纹理贴图的过程中不存在纹理扭曲形变的现象，即在将纹理三角形映射至3D三角形上时，仅需执行刚体变换（旋转、平移操作）。现在，我们就可以把纹理看作是一张贴纸，通过拾取、平移以及旋转等手段，将它贴在3D三角形上。图19.4展示了纹理坐标系的坐标轴相对于3D三角形的位置关系：这些坐标轴与三角形相切，并与三角形处于同一平面。所以，三角形的纹理坐标势必也就相对于此纹理坐标系而定。再结合三角形的平面法线$\boldsymbol{N}$，我们便获得了一个位于三角形所在平面内的3D **TBN基**（TBN-basis），它常被称为纹理空间（texture space）或切线空间（tangent space，也有译作正切空间、切空间等）[1]。值得注意的是，切线空间通常会随不同的三角形而发生改变（见图19.5）。

现在把目光转回图19.3，此法线图中的法向量是相对于纹理空间而定义的。可是，我们所用的光源都被定义在世界空间之中。为了实现光照效果，法向量与光源必须位于同一空间内。因此，当前的首要任务就是使三角形顶点所在的物体空间坐标系（object space coordinate system）与切线空间坐标系建立联系。一旦这些顶点位于物体空间之中，我们就能通过世界矩阵将它们从物体空间变换到世界空间（在下一节中会讨论相关细节）。设纹理坐标分别为(u_0, v_0)、(u_1, v_1)、(u_2, v_2)的顶点$\boldsymbol{v}_0$、$\boldsymbol{v}_1$和$\boldsymbol{v}_2$在相对于纹理坐标系坐标轴（即切向量$\boldsymbol{T}$与切向量$\boldsymbol{B}$）构成

的纹理平面内定义了一个三角形。设$\boldsymbol{e}_0=\boldsymbol{v}_1-\boldsymbol{v}_0$且$\boldsymbol{e}_1=\boldsymbol{v}_2-\boldsymbol{v}_0$为3D三角形的两个边向量，它们所对应的纹理三角形边向量则分别为$(\Delta u_0,\ \Delta v_0)=(u_1-u_0,\ v_1-v_0)$与$(\Delta u_1,\ \Delta v_1)=(u_2-u_0,\ v_2-v_0)$。由图19.4可以看出它们的关系：

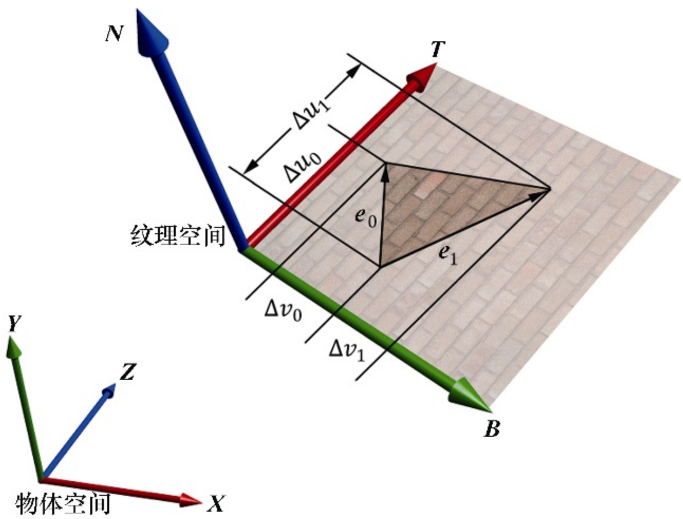


图19.4　一个三角形纹理空间与物体空间的关系。3D切向量$\boldsymbol{T}$指向纹理坐标系u轴的正方向，而3D切向量$\boldsymbol{B}$则指向纹理坐标系v轴的正方向

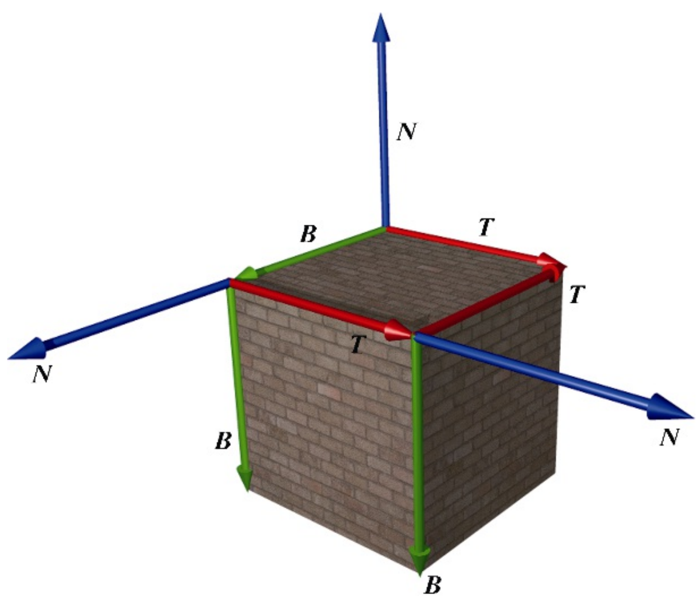


图19.5　此立方体诸面的纹理空间都是各不相同的

$$\begin{aligned}\boldsymbol{e}_0 &= \Delta u_0\boldsymbol{T} + \Delta v_0\boldsymbol{B}\\ \boldsymbol{e}_1 &= \Delta u_1\boldsymbol{T} + \Delta v_1\boldsymbol{B}\end{aligned}$$

用相对于物体空间的坐标来表示这些向量，就能得到其矩阵方程：

$$\begin{bmatrix} e_{0,x} & e_{0,y} & e_{0,z} \\ e_{1,x} & e_{1,y} & e_{1,z} \end{bmatrix} = \begin{bmatrix} \Delta u_0 & \Delta v_0 \\ \Delta u_1 & \Delta v_1 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

注意，我们已经知道了三角形顶点的物体空间坐标，当然也就能求出边向量的物体空间坐标。因此，矩阵

$$\begin{bmatrix} e_{0,x} & e_{0,y} & e_{0,z} \\ e_{1,x} & e_{1,y} & e_{1,z} \end{bmatrix}$$

是已知的。同理，由于我们也知道了纹理坐标，因而矩阵

$$\begin{bmatrix} \Delta u_0 & \Delta v_0 \\ \Delta u_1 & \Delta v_1 \end{bmatrix}$$

亦可知晓。现在就来求出切向量$\boldsymbol{T}$与切向量$\boldsymbol{B}$的物体空间坐标：

$$\begin{aligned}\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} &= \begin{bmatrix} \Delta u_0 & \Delta v_0 \\ \Delta u_1 & \Delta v_1 \end{bmatrix}^{-1} \begin{bmatrix} e_{0,x} & e_{0,y} & e_{0,z} \\ e_{1,x} & e_{1,y} & e_{1,z} \end{bmatrix}\\ &= \frac{1}{\Delta u_0 \Delta v_1 - \Delta v_0 \Delta u_1} \begin{bmatrix} \Delta v_1 & -\Delta v_0 \\ -\Delta u_1 & \Delta u_0 \end{bmatrix} \begin{bmatrix} e_{0,x} & e_{0,y} & e_{0,z} \\ e_{1,x} & e_{1,y} & e_{1,z} \end{bmatrix}\end{aligned}$$

在上述推导过程中，我们使用了下列关于逆矩阵的性质，假设有矩阵$\boldsymbol{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$，则有：

$$\boldsymbol{A}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

要知道，切向量$\boldsymbol{T}$与切向量$\boldsymbol{B}$在物体空间中一般均不是单位长度。

而且，如果纹理发生了扭曲形变，那么这两个向量也将不再互为正交规范化向量。

按照惯例，$\boldsymbol{T}$、$\boldsymbol{B}$、$\boldsymbol{N}$三个向量通常分别被称为切线（tangent，也有为了区分副切线而称为主切线）、副法线[2]（binormal，亦有译作次法线。或bitangent，副切线）以及法线（normal）。

19.4　顶点切线空间

在上一节中，我们推导出了任意三角形所对应的切线空间。但是，如果在进行法线贴图时使用了这种纹理空间，则最终得到的效果会呈现明显的三角形划分痕迹，这是因为切线空间必定位于对应三角形的所在平面，以致贴图面不够圆滑，直来直去全是棱角。因此，我们要在每个顶点处指定切向量，并重施求平均值的故技，令顶点法线更趋于平滑的表面，化粗糙为神奇（可回顾8.2节进行对比）。

1．通过计算网格中共用顶点$\boldsymbol{v}$的每个三角形的切向量平均值，便可以求出网格中任意顶点$\boldsymbol{v}$处的切向量$\boldsymbol{T}$。

2．通过计算网格中共用顶点$\boldsymbol{v}$的每个三角形的副切向量平均值，便能够求此网格中顶点$\boldsymbol{v}$处的副切向量$\boldsymbol{B}$。

一般来讲，在计算完上述平均值之后，往往还需要对TBN基进行正交规范化处理，使这3个向量相互正交且均具有单位长度。这通常是以格拉姆—施密特过程（Gram-Schmidt procedure）来实现的。可以在网络上找到为任意三角形网格构建逐顶点切线空间的相关代码[3]。

在我们当前的系统中，并不会把副切向量$\boldsymbol{B}$直接存于内存中。而是在用到$\boldsymbol{B}$时以$\boldsymbol{B}=\boldsymbol{N}\times\boldsymbol{T}$来求取此向量，公式中的$\boldsymbol{N}$即顶点法线的平均值（平均顶点法线）。因此，我们所定义的顶点结构体如下：

```
struct Vertex
{
```

```
  XMFLOAT3 Pos;
  XMFLOAT3 Normal;
  XMFLOAT2 TexC;
  XMFLOAT3 TangentU;
};
```

前文曾提到，我们在程序中是通过GeometryGenerator函数计算纹理空间中坐标轴u所对应的切向量$\boldsymbol{T}$来生成网格的（详见7.4.1节）。对于立方体和栅格这两种网格来说，指定其每个顶点处切向量$\boldsymbol{T}$的物体空间坐标并不是件难事（见图19.5）。而对于圆柱体与球体来讲，为了求出它们每个顶点处的切向量$\boldsymbol{T}$，我们就要构建圆柱体或球体其具有两个变量的向量值函数$\boldsymbol{P}(u,v)$并计算出$\partial \boldsymbol{p}/\partial u$，其中的参数$u$也常被用作纹理坐标$u$。

19.5 在切线空间与物体空间之间进行转换

至此，我们确定了网格中每个顶点处的正交规范化TBN基，而且也已求出了TBN三向量相对于网格物体空间的坐标。也就是说，我们已经掌握了TBN基相对于物体空间坐标系的坐标，并可通过下列矩阵将坐标由切线空间变换至物体空间：

$$\boldsymbol{M}_{object}=\begin{bmatrix}T_x & T_y & T_z\\ B_x & B_y & B_z\\ N_x & N_y & N_z\end{bmatrix}$$

由于该矩阵是正交矩阵，所以其逆矩阵就是它的转置矩阵。因此，由物体空间转换到切线空间的坐标变换矩阵为：

$$\boldsymbol{M}_{tangent}=\boldsymbol{M}_{object}^{-1}=\boldsymbol{M}_{object}^{\mathrm{T}}=\begin{bmatrix}T_x & B_x & N_x\\ T_y & B_y & N_y\\ T_z & B_z & N_z\end{bmatrix}$$

在我们所编写的着色器程序中，为了计算光照会把法向量从切线空间变换到世界空间。对此，一种可行方法是首先将法线自切线空间变换至物体空间，而后再以世界矩阵把它从物体空间变换到世界空间：

$$\boldsymbol{n}_{world}=(\boldsymbol{n}_{tangent}\boldsymbol{M}_{object})\boldsymbol{M}_{world}$$

然而，由于矩阵乘法满足结合律，因此可以将公式变为：

$$\boldsymbol{n}_{world}=\boldsymbol{n}_{tangent}(\boldsymbol{M}_{object}\boldsymbol{M}_{world})$$

注意到

$$\boldsymbol{M}_{object}\boldsymbol{M}_{world} = \begin{bmatrix} \leftarrow & \boldsymbol{T} & \rightarrow \\ \leftarrow & \boldsymbol{B} & \rightarrow \\ \leftarrow & \boldsymbol{N} & \rightarrow \end{bmatrix} \boldsymbol{M}_{world} = \begin{bmatrix} \leftarrow & \boldsymbol{T}' & \rightarrow \\ \leftarrow & \boldsymbol{B}' & \rightarrow \\ \leftarrow & \boldsymbol{N}' & \rightarrow \end{bmatrix} = \begin{bmatrix} T_x' & T_y' & T_z' \\ B_x' & B_y' & B_z' \\ N_x' & N_y' & N_z' \end{bmatrix}$$

其中$\boldsymbol{T}' = \boldsymbol{T} \cdot \boldsymbol{M}_{world}$、$\boldsymbol{B}' = \boldsymbol{B} \cdot \boldsymbol{M}_{world}$以及$\boldsymbol{N}' = \boldsymbol{N} \cdot \boldsymbol{M}_{world}$。因此，要从切线空间直接变换到世界空间，我们仅需用世界坐标来表示切线基（tangent basis）即可，这可以通过将TBN基从物体空间坐标变换到世界空间坐标来实现。

我们所关心的仅是向量的变换（而非点的变换），因此用3×3矩阵来表示即可。本书前面曾讲过，仿射矩阵的第4行是用于执行平移变换的，但我们却不能平移向量。

19.6　法线贴图的着色器代码

法线贴图的流程大致如下。

1．通过艺术加工工具或图像处理程序来创造预定的法线图，并将它存于图像文件之中。在应用程序初始化期间以这些图像文件来创建2D纹理。

2．针对每一个三角形，计算其切向量$\boldsymbol{T}$。通过对网格中共享顶点$\boldsymbol{v}$的所有三角形的切向量求取平均值，就可以获取此网格中每个顶点$\boldsymbol{v}$处的切向量（在演示程序中，由于使用的是简单的几何图形，因此可以直接指定出相应的切向量。但是，如果处理的是由3D建模程序创建的不规则形状的三角形网格，那么就需要按上述方法来计算切向量的平均值）。

3．在顶点着色器中，将顶点法线与切向量变换到世界空间，并将结果输出到像素着色器。

4．通过插值切向量与插值法向量来构建三角形表面每个像素点处的TBN基，再以此TBN基将从法线图中采集的法向量由切线空间变换到世界空间。这样一来，我们就拥有了取自法线图的世界空间法向量，并可将它用于往常的光照计算。

为便于法线贴图的实现，我们在Common.hlsl文件中加入了以下函数。

```
//---------------------------------------------------------------------------------
// 将一个法线图样本变换至世界空间
//---------------------------------------------------------------------------------
float3 NormalSampleToWorldSpace(float3 normalMapSample,
                float3 unitNormalW,
                float3 tangentW)
{
    // 将每个坐标分量由范围[0,1]解压至[-1,1]区间
    float3 normalT = 2.0f*normalMapSample - 1.0f;

    // 构建正交规范基
    float3 N = unitNormalW;
    float3 T = normalize(tangentW - dot(tangentW, N)*N);
    float3 B = cross(N, T);

    float3x3 TBN = float3x3(T, B, N);

    // 将法线图样本从切线空间变换到世界空间
    float3 bumpedNormalW = mul(normalT, TBN);

    return bumpedNormalW;
}
```

此函数可以在像素着色器中按下列方式调用：

```
float3 normalMapSample = gNormalMaps.Sample(samLinear,pin.Tex).rgb;
float3 bumpedNormalW = NormalSampleToWorldSpace(normalMapSample, pin.Norma
lW, pin. TangentW);
```

有两行代码可能不太容易理解：

```
float3 N = unitNormalW;
float3 T = normalize(tangentW - dot(tangentW, N)*N);
```

插值完成后，切向量与法向量可能会变为非正交规范向量。以上两行代码通过使$\boldsymbol{T}$减去其$\boldsymbol{N}$方向上的分量（投影），再对结果进行规范化处理，从而使$\boldsymbol{T}$成为规范化向量且正交于$\boldsymbol{N}$（见图19.6）。注意，这里假设unitNormalW为规范化向量。

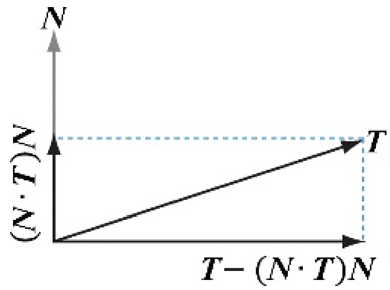


图19.6　由于$\|\boldsymbol{N}\| = 1$，$\mathrm{proj}_{N}(\boldsymbol{T}) = (\boldsymbol{T}\cdot\boldsymbol{N})\boldsymbol{N}$，所以切线$\boldsymbol{T}$的分量$\boldsymbol{T} - \mathrm{proj}_{N}(\boldsymbol{T})$正交于法线$\boldsymbol{N}$

只要获取了法线图中的法线（也称“bumped normal”，常直译作凹凸法线，即方向不一的法线，可令光照表现出物体表面凹凸不平的效果），即可将它运用于法向量所参与的一切后续计算（如光照、立方体贴图）。完整的法线贴图效果实现如下，其中与法线贴图相关的部分都已用黑体字标出。

```
//*********************************************************************
// Default.hlsl的作者为Frank Luna (C) 2015 版权所有
//*********************************************************************

// 默认的光源数量
#ifndef NUM_DIR_LIGHTS
  #define NUM_DIR_LIGHTS 3
#endif

#ifndef NUM_POINT_LIGHTS
  #define NUM_POINT_LIGHTS 0
#endif

#ifndef NUM_SPOT_LIGHTS
  #define NUM_SPOT_LIGHTS 0
#endif

// 包含公用的HLSL代码
#include "Common.hlpsl"

struct VertexIn
{
  float3 PosL  : POSITION;
```

```
  float3 NormalL : NORMAL;
  float2 TexC    : TEXCOORD;
  float3 TangentU : TANGENT;
};

struct VertexOut
{
  float4 PosH    : SV_POSITION;
  float3 PosW    : POSITION;
  float3 NormalW : NORMAL;
  float3 TangentW : TANGENT;
  float2 TexC    : TEXCOORD;
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout = (VertexOut)0.0f;

  // 获取材质数据
  MaterialData matData = gMaterialData[gMaterialIndex];

  // 把顶点变换到世界空间
  float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
  vout.PosW = posW.xyz;

  // 假设这里执行的是等比缩放，否则就需要使用世界矩阵的逆转置矩阵进行变换
  vout.NormalW = mul(vin.NormalL, (float3x3)gWorld);

  vout.TangentW = mul(vin.TangentU, (float3x3)gWorld);

  // 将顶点变换到齐次裁剪空间
  vout.PosH = mul(posW, gViewProj);

  // 为三角形插值而输出顶点属性
  float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
  vout.TexC = mul(texC, matData.MatTransform).xy;

  return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
  // 获取材质数据
  MaterialData matData = gMaterialData[gMaterialIndex];
  float4 diffuseAlbedo = matData.DiffuseAlbedo;
  float3 fresnelR0 = matData.FresnelR0;
  float roughness = matData.Roughness;
  uint diffuseMapIndex = matData.DiffuseMapIndex;
```

```
  uint normalMapIndex = matData.NormalMapIndex;

  // 对法线插值可能使它非规范化，因此要再次对它进行规范化处理
  pin.NormalW = normalize(pin.NormalW);

  float4 normalMapSample = gTextureMaps[normalMapIndex].Sample(
    gsamAnisotropicWrap, pin.TexC);
  float3 bumpedNormalW = NormalSampleToWorldSpace(
    normalMapSample.rgb, pin.NormalW, pin.TangentW);

  // 去掉下列注释则禁用法线贴图
  //bumpedNormalW = pin.NormalW;

  // 动态查找数组中的纹理
  diffuseAlbedo *= gTextureMaps[diffuseMapIndex].Sample(
    gsamAnisotropicWrap, pin.TexC);

  // 由表面上某点指向观察点的向量
  float3 toEyeW = normalize(gEyePosW - pin.PosW);

  // 光照项
  float4 ambient = gAmbientLight*diffuseAlbedo;

  // Alpha通道存储的是逐像素级别上的光泽度
  const float shininess = (1.0f - roughness) * normalMapSample.a;
  Material mat = { diffuseAlbedo, fresnelR0, shininess };
  float3 shadowFactor = 1.0f;
  float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
    bumpedNormalW, toEyeW, shadowFactor);

  float4 litColor = ambient + directLight;

  // 利用法线图中的法线计算镜面反射
  float3 r = reflect(-toEyeW, bumpedNormalW);
  float4 reflectionColor = gCubeMap.Sample(gsamLinearWrap, r);
  float3 fresnelFactor = SchlickFresnel(fresnelR0, bumpedNormalW, r);
  litColor.rgb += shininess * fresnelFactor * reflectionColor.rgb;

  // 从漫反射反照率中获取alpha 值的常规方法
  litColor.a = diffuseAlbedo.a;

  return litColor;
}
```

可以看出，那些名为“凹凸法线（bumped normal）”的向量既可用于

光照计算，又能用在以环境图模拟反射的反射效果计算当中。另外，我们在法线图的alpha通道存储的是光泽度掩码（shininess mask），它控制着逐像素级别上物体表面的光泽度（见图19.7）。

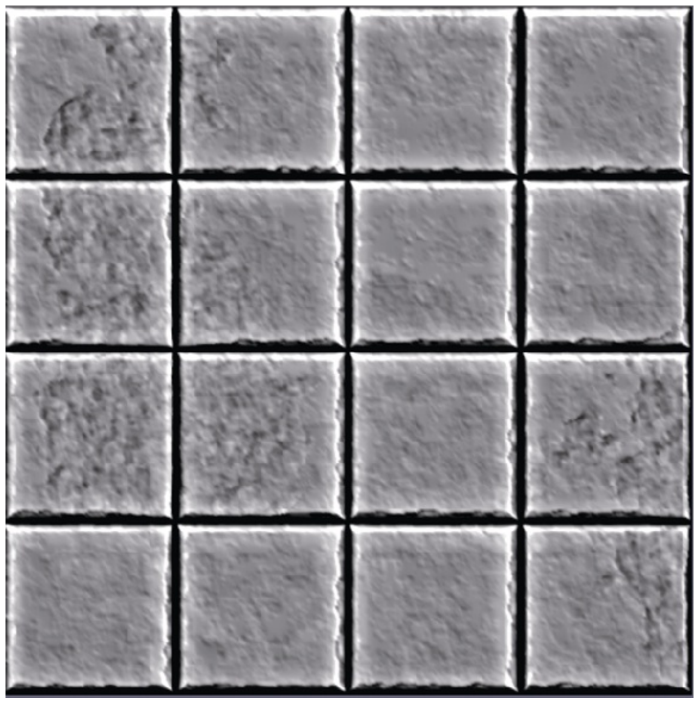


图19.7　随书所附DVD中tile_nmap.dds图像所存的alpha通道。该alpha通道中的数据表示的是物体表面的光泽度。白色部分的光泽度值为1.0，黑色部分的光泽度值则为0.0。这样，我们就获得了一种用于逐像素控制光泽度的材质属性

19.7　小结

1．法线贴图的策略是将法线图映射到多边形之上。这样一来即可得到每个像素的对应法线，通过它便可以捕捉到物体表面上的凸起、凹陷、裂痕等更为丰富的细节。因此，我们就能利用这些取自法线图中的逐像素法线来代替插值顶点法线进行光照计算。

2．法线图本质上就是一种纹理，但它的每个纹素中存储的并非RGB颜色数据，在其红色、绿色以及蓝色3种分量中分别存储的是经过压缩的x、y、z法线坐标。我们可以利用各种工具来生成法线图。

3．法线图中的法线坐标相对于纹理空间坐标系而定。因此，为了实现光照计算，我们需要将法线从纹理空间变换至世界空间，以令光源与法线位于同一坐标系中。构建在每个顶点处的TBN基则用于协助从纹理空间到世界空间的变换。

19.8　练习

1．下载NVIDIA公司制作的法线图插件，并用它来创建几幅不同的法线图来熟悉它的使用方法。试着将生成的法线图应用到本章的演示程序当中。

2．下载CrazyBump程序的试用版。用此程序来加载彩色图像，并生成法线图与位移图（displacement map，见第5题）来熟悉此程序的用法。最后尝试将这两种图用于本章的应用程序之中。

3．如果我们对纹理应用了旋转变换，那么还需要旋转相应的切线空间坐标系。试解释为什么要这样做。事实上，这就意味着我们需要在世界空间中使切线$\boldsymbol{T}$绕着法线$\boldsymbol{N}$旋转，这会涉及开销较大的三角函数运算（更准确地说，要执行的是关于任意轴$\boldsymbol{N}$的旋转变换）。而另一种解决方案则是先将$\boldsymbol{T}$由世界空间变换到切线空间，这时，我们就可以利用纹理变换矩阵来直接对$\boldsymbol{T}$进行旋转操作，而后再将其变换回世界空间。

4．若不希望在世界空间中进行光照计算，我们可以将观察向量与光向量都由世界空间变换到切线空间，并将所有的光照运算都移至切线空间开展。试着修改法线贴图着色器，在切线空间中进行光照计算。

5．位移贴图（displacement mapping，也有译作置换贴图等）的思路是引入一种名为高度图（heightmap）的新品贴图来描述物体表面的凹凸不平。此图常与硬件曲面细分配合使用，它指示了新增加的顶点在法向量方向上的偏移量，以此来为网格增添几何细节。位移贴图可用于

实现海浪效果，方法是在一个平坦的顶点栅格上，以不同的速度和方向“滚动”两幅（或更多）高度图。我们在每个栅格的顶点处对这些高度图进行采样，并将同一顶点处的高度值加在一起，所得到的高度值和便是某时刻此实例在该顶点处的高度（即y坐标）。通过滚动高度图，水波就能连续而此起彼伏地自然流动起来，同时营造出一种“滚滚长江东逝水”的视觉效果（见图19.8）。设置这个练习的目标为通过两幅可以下载到的海浪高度图（以及对应的法线图，见图19.9）[4]来实现海浪的效果。为了展现出更好的波浪效果，这里给出以下几点提示。

（a）使用不同的高度图：以一个高波幅（high amplitude）的高度图来模拟波面宽阔的低频波浪，而以另一个低波幅（low amplitude）的高度图来模拟波面细碎却波涛汹涌的高频波浪。因此，我们就需要针对这两种高度图采用两组不同纹理坐标，以及分别运用两种不同的纹理变换。

（b）所用的法线图纹理应当比高度图纹理的数量要多。高度图指定了波浪的形状，而法线图则负责波浪中像素的照明工作。法线图应当与高度图一同随时间流逝而进行平移，并以不同方向的移动给人以波浪随机翻腾隐现的幻觉。这两种法线可以按下列方式搭配在一起运用：

```
float3 normalMapSample0 = gNormalMap0.Sample(samLinear, pin.WaveNormalTex0
).rgb;
float3 bumpedNormalW0 = NormalSampleToWorldSpace(
normalMapSample0, pin.NormalW, pin.TangentW);

float3 normalMapSample1 = gNormalMap1.Sample(samLinear, pin.WaveNormalTex1
).rgb;
float3 bumpedNormalW1 = NormalSampleToWorldSpace(
normalMapSample1, pin.NormalW, pin.TangentW);
```

```
float3 bumpedNormalW = normalize(bumpedNormalW0 + bumpedNormalW1);
```

（c）修改波浪的材质，使它更接近于海洋的蓝色，并且保留来自环境图的部分反射。

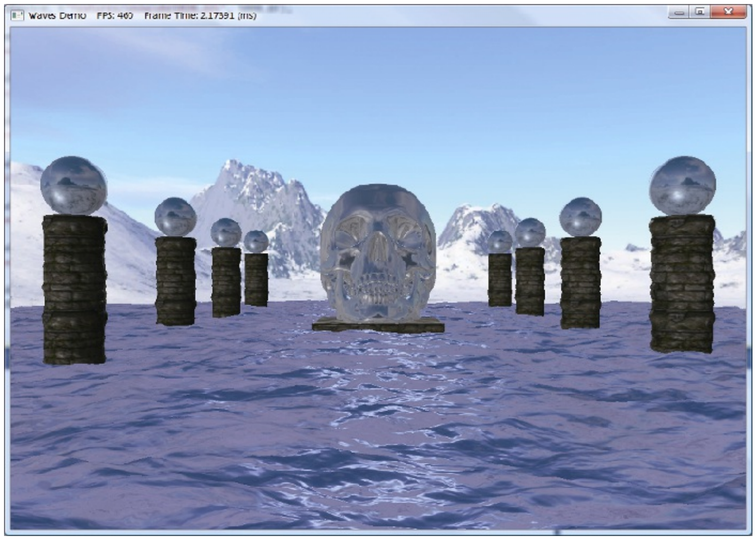


图19.8　用高度图、法线图以及环境贴图所模拟出的海洋波浪效果

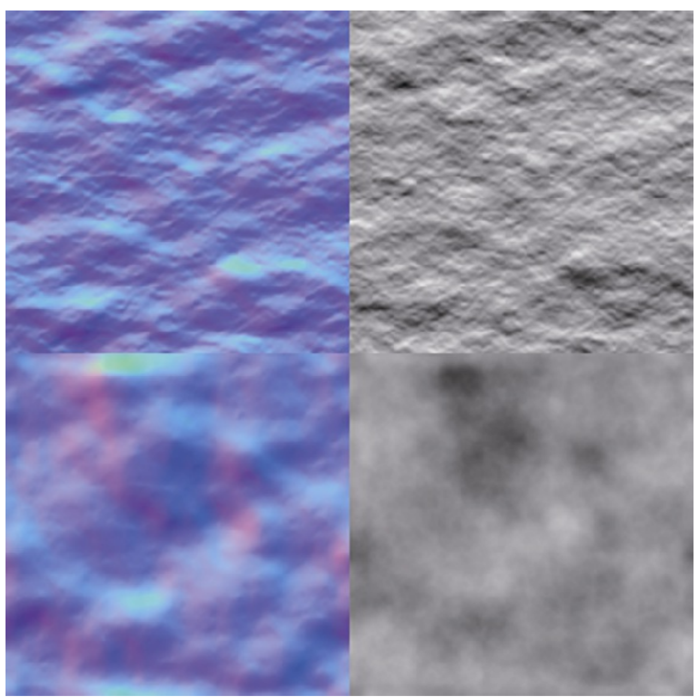


图19.9　上面的一组图分别是用于绘制高频细碎波浪的法线图和高度图，

下面的一组图分别是用于渲染低频宽阔波浪的法线图与高度图

---

[1] 纹理坐标系是一种2D坐标系，而这里称为“纹理空间”的切线空间则是一种3D坐标系。为了避免发生歧义，建议不要将切线空间称为纹理空间，这里称纹理空间纯属按原文翻译。

[2] 严格来讲，在讨论切线空间的时候，切线空间是由两个切向量和一个法向量来确定的（从图中19.4中也可以看出）。因此，这三个向量应分别名为“（主）切线”（tangent）、“副（次）切线”（bitangent）以及“法线”（normal）。可参见“Bitangent versus Binormal”或叛逆者的《Binormal，还是Bitangent，这是个问题》。

[3] 原文名为《Computing Tangent Space Basis Vectors for an Arbitrary Mesh》。感谢博主“码瘾少年·麒麟子”对原文进行了翻译。

[4] 本书配套网站上的d3d11CodeSet3.zip文件，纹理路径为d3d11CodeSet3\SelectedCodeSolutions\DisplacementMappedWaves\Textures。

第20章　阴影贴图

阴影既暗示着光源相对于观察者的位置关系，也从侧面传达了场景中各物体之间的相对位置。本章将介绍一种阴影贴图的基本算法，这是一种在游戏以及3D应用程序中模拟动态阴影的常见方法。对于读者眼前这样一本介绍性的入门书籍来说，我们仅关注于这个基本的阴影贴图算法。而像级联阴影贴图（cascading shadow map）[Engel06]这种效果更佳却也更为复杂的阴影技术，实则都是由这基本的阴影贴图算法扩展而成的。

学习目标：

1．探究基本的阴影贴图算法。

2．学习投影纹理贴图的工作原理。

3．计算正交投影。

4．了解阴影图的走样问题并学习修正该问题的常用策略。

20.1 渲染场景深度

阴影贴图（shadow mapping，也有译作阴影映射）算法依赖于以光源的视角渲染场景深度（scene depth，即场景中的深度信息）——从本质上来讲，这其实就是一种变相的“渲染到纹理”技术，本书曾在13.7.2节中对后者进行了第一次阐述。从“渲染场景深度”这个名字就可以看出，构建此深度缓冲区要从光源的视角着手。待以光源视角渲染场景深度之后，我们就能知晓离光源最近的像素片段，即那些不在阴影范围之中的像素片段。本节将考察一个名为ShadowMap的工具类，它所存储的是以光源为视角而得到的场景深度数据。其实此工具类就是简单地封装了渲染阴影会用到的一个深度/模板缓冲区、一个视口以及多个视图，而用于阴影贴图的那个深度/模板缓冲区也被称为阴影图（shadow map）。

```
class ShadowMap
{
public:
  ShadowMap(ID3D12Device* device,
    UINT width, UINT height);

  ShadowMap(const ShadowMap& rhs)=delete;
  ShadowMap& operator=(const ShadowMap& rhs)=delete;
  ~ShadowMap()=default;

  UINT Width()const;
  UINT Height()const;
  ID3D12Resource* Resource();
  CD3DX12_GPU_DESCRIPTOR_HANDLE Srv()const;
  CD3DX12_CPU_DESCRIPTOR_HANDLE Dsv()const;

  D3D12_VIEWPORT Viewport()const;
  D3D12_RECT ScissorRect()const;
```

```
  void BuildDescriptors(
    CD3DX12_CPU_DESCRIPTOR_HANDLE hCpuSrv,
    CD3DX12_GPU_DESCRIPTOR_HANDLE hGpuSrv,
    CD3DX12_CPU_DESCRIPTOR_HANDLE hCpuDsv);

  void OnResize(UINT newWidth, UINT newHeight);

private:
  void BuildDescriptors();
  void BuildResource();

private:

  ID3D12Device* md3dDevice = nullptr;

  D3D12_VIEWPORT mViewport;
  D3D12_RECT mScissorRect;

  UINT mWidth = 0;
  UINT mHeight = 0;
  DXGI_FORMAT mFormat = DXGI_FORMAT_R24G8_TYPELESS;

  CD3DX12_CPU_DESCRIPTOR_HANDLE mhCpuSrv;
  CD3DX12_GPU_DESCRIPTOR_HANDLE mhGpuSrv;
  CD3DX12_CPU_DESCRIPTOR_HANDLE mhCpuDsv;

  Microsoft::WRL::ComPtr<ID3D12Resource> mShadowMap = nullptr;
};
```

该工具类的构造函数会根据指定的尺寸和视口来创建纹理。阴影图的分辨率（resolution，也有译作解析度）会直接影响阴影效果的质量，但是在提升该分辨率的同时，渲染也将产生更多的开销并占用更多的内存。

```
ShadowMap::ShadowMap(ID3D12Device* device, UINT width, UINT height)
{
  md3dDevice = device;

  mWidth = width;
  mHeight = height;

  mViewport = { 0.0f, 0.0f, (float)width, (float)height, 0.0f, 1.0f };
  mScissorRect = { 0, 0, (int)width, (int)height };
```

```
  BuildResource();
}

void ShadowMap::BuildResource()
{
  D3D12_RESOURCE_DESC texDesc;
  ZeroMemory(&texDesc, sizeof(D3D12_RESOURCE_DESC));
  texDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
  texDesc.Alignment = 0;
  texDesc.Width = mWidth;
  texDesc.Height = mHeight;
  texDesc.DepthOrArraySize = 1;
  texDesc.MipLevels = 1;
  texDesc.Format = mFormat;
  texDesc.SampleDesc.Count = 1;
  texDesc.SampleDesc.Quality = 0;
  texDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
  texDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL;

  D3D12_CLEAR_VALUE optClear;
  optClear.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
  optClear.DepthStencil.Depth = 1.0f;
  optClear.DepthStencil.Stencil = 0;

  ThrowIfFailed(md3dDevice->CreateCommittedResource(
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
    D3D12_HEAP_FLAG_NONE,
    &texDesc,
    D3D12_RESOURCE_STATE_GENERIC_READ,
    &optClear,
    IID_PPV_ARGS(&mShadowMap)));
}
```

我们即将认识到：阴影贴图算法需要执行两个渲染过程（render pass）。在第一次渲染过程中，我们要以光源的视角将场景深度数据渲染至阴影图中（后文中有时称之为深度绘制过程，depth pass）；而在第二次渲染过程中则像往常那样，以“玩家”的摄像机视角将场景渲染至后台缓冲区之内，但为了实现阴影算法，此时应以阴影图作为着色器的输入之一。我们为访问着色器资源及其视图而提供了以下方法。

```
ID3D12Resource* ShadowMap::Resource()
{
  return mShadowMap.Get();
}

CD3DX12_GPU_DESCRIPTOR_HANDLE ShadowMap::Srv()const
{
   return mhGpuSrv;
}

CD3DX12_CPU_DESCRIPTOR_HANDLE ShadowMap::Dsv()const
{
   return mhCpuDsv;
}
```

20.2　正交投影

我们此前一直使用的是透视投影（perspective projection）。透视投影的关键特性在于，物体离观察点越远，它们就会显得越小（即近大远小）。这与我们在现实生活中对物体的感官是一致的。除此之外，还有一种名为正交投影（orthographic projection，也有译作正投影）的投影类型。这种投影主要运用于3D科学或工程应用之中，即需要平行线在投射之后继续保持平行的情景（而不是像透视投影那样交于灭点）。因此，正交投影只适用于模拟平行光所生成的阴影。沿着观察空间z轴的正方向看去，正交投影的视景体（viewing volume）是宽度为w、高度为h、近平面为n、远平面为f的对齐于观察空间坐标轴的长方体（见图20.1）。这些数据定义了相对于观察空间坐标系的长方体视景体。

使用正交投影时，其投影线均平行于观察空间的z轴（见图20.2）。因此我们可以看到，顶点(x, y, z)的2D投影为(x, y)。

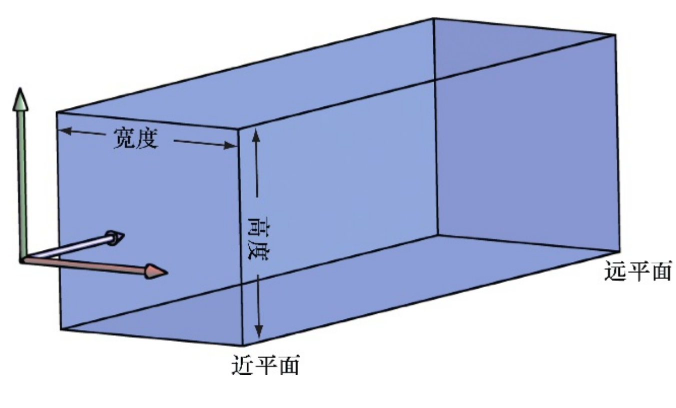


图20.1　正交视景体是一个轴对齐于观察坐标系主轴的长方体

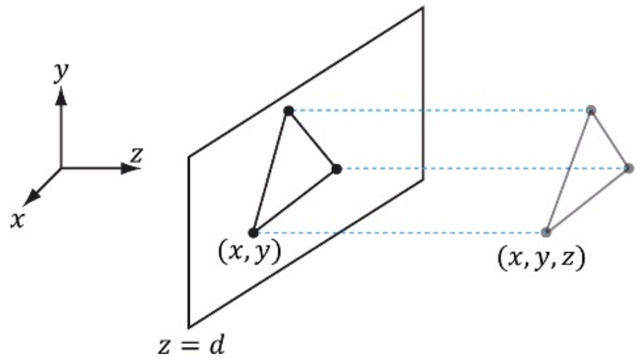


图20.2　诸点在投影平面上的正交投影。正交投影的投影线均平行于观察空间的z轴

与透视投影一样，我们既希望保留正交投影中的深度信息，又希望采用规格化设备坐标。为了将正交投影视景体从观察空间变换到NDC（规格化设备坐标）空间，我们就需要通过缩放以及平移操作来将观察空间中的视景体$\left[-\frac{w}{2},\frac{w}{2}\right]\times\left[-\frac{h}{2},\frac{h}{2}\right]\times[n,f]$映射为NDC空间的视景体$[-1,\ 1]\times[-1,\ 1]\times[0,\ 1]$。把上述关键坐标逐个进行对比变换即可确定该映射。对于前两对坐标来说，通过观察区间的差别，即可借助缩放因子方便地将它们变换到NDC空间：

$$\frac{2}{w}\cdot\left[-\frac{w}{2},\frac{w}{2}\right]=[-1,1]$$
$$\frac{2}{h}\cdot\left[-\frac{h}{2},\frac{h}{2}\right]=[-1,1]$$

而针对第3对坐标，则需要实现$[n,f]\to[0,\ 1]$这一映射。在此，我们将其表示为方程$g(z)=az+b$（即一次缩放与平移变换）。由于$g(n)=0$且$g(f)=1$，我们便可以解出a与b：

$$an+b=0$$
$$af+b=1$$

通过第一个方程可知$b = -an$。将它代入第二个方程，则有：

$$af - an = 1$$
$$a = \frac{1}{f-n}$$

所以：

$$-\frac{n}{f-n} = b$$

因此，

$$g(z) = \frac{z}{f-n} - \frac{n}{f-n}$$

读者不妨绘制出当变量n与f满足$f > n$时，函数$g(z)$在$[n, f]$区间内的图像。

最后，我们可得到将观察空间坐标(x, y, z)转换到NDC空间坐标(x', y', z')的正交变换：

$$x' = \frac{2}{w}x$$
$$y' = \frac{2}{h}y$$
$$z' = \frac{z}{f-n} - \frac{n}{f-n}$$

或者用矩阵表示为：

$$[x',\ y',\ z',\ 1] = [x,\ y,\ z,\ 1]\ \begin{bmatrix} \frac{2}{w} & 0 & 0 & 0 \\ 0 & \frac{2}{h} & 0 & 0 \\ 0 & 0 & \frac{1}{f-n} & 0 \\ 0 & 0 & \frac{n}{n-f} & 1 \end{bmatrix}$$

上述等式中的4×4矩阵即为正交投影矩阵（orthographic projection matrix）。

回顾透视投影变换可以知道，我们不得不将这个变换过程分为两个部分：一个是通过投影矩阵描述的线性部分，另一个则是通过除以w分量来描述的非线性部分。相比之下，正交投影变换则完全是一种线性变换——因为整个过程都不需要除以w分量。将原坐标乘以正交投影矩阵便可以直接将其变换为NDC坐标。

20.3　投影纹理坐标

投影纹理贴图（projective texturing，也有译作投影纹理映射、投影贴图等）技术能够将纹理投射到任意形状的几何体上，又因为其原理与投影机的工作方式比较相似，故而得名。图20.3展示的是投影纹理贴图的一个示例。

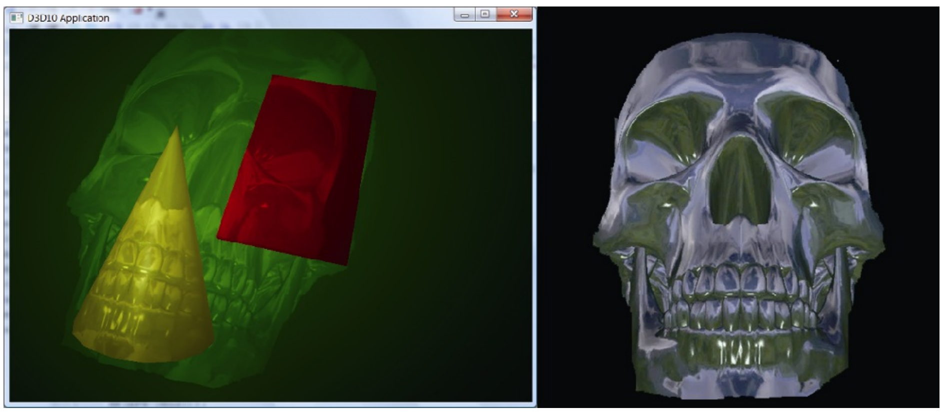


图20.3　通过投影纹理贴图技术将右图中的骷髅头纹理投射到左图场景中的多个几何体上

投影纹理贴图完美地模拟了投影机投射光线的过程。而在20.4节中我们将看到，事实上它也充当着阴影贴图过程中的一个关键环节。

投影纹理贴图的关键在于为每个像素生成对应的纹理坐标，从视觉上给人一种纹理被投射到几何体上的感觉。我们将生成的这种纹理坐标称为投影纹理坐标（projective texture coordinate）。

从图20.4可以看出，纹理坐标(u, v)指定了应当被投射到3D点$\boldsymbol{p}$上的纹素。又由于坐标(u, v)相对于投影窗口中的纹理空间坐标系，精确地指出了该投影窗口中点$\boldsymbol{p}$的投影。因此，生成投影纹理坐标的过程可分为

如下步骤。

1．将点$\boldsymbol{p}$投影至光源的投影窗口，并将其坐标变换到NDC空间。

2．将投影坐标从NDC空间变换到纹理空间，以此将它们转换为纹理坐标。

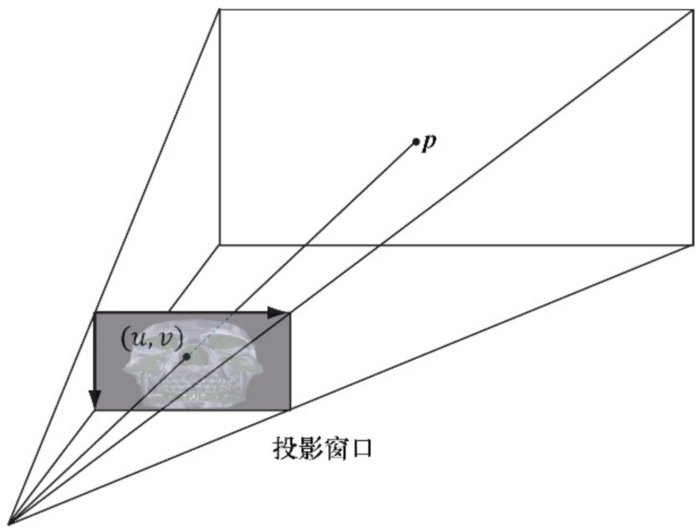


图20.4　根据光源射向点$\boldsymbol{p}$的投影线，我们便能通过相对于投影窗口中纹理空间的坐标(u, v)来确定点$\boldsymbol{p}$在投影窗口中的纹素

通过将投影机的光源装置作为摄像机便可以实现步骤1。我们为此光源装置定义了观察矩阵$\boldsymbol{V}$以及投影矩阵$\boldsymbol{P}$。从本质上来讲，这两个矩阵分别定义了光源装置位于世界空间中的位置、朝向以及视锥体。矩阵$\boldsymbol{V}$能够将坐标从世界空间变换到光源装置的坐标系（此时，以投影机的光源作为“摄像机”）。只要是相对于光源坐标系的顶点坐标，经过投影矩阵变换以及齐次除法，我们都可以将其投影到光源的投影平面上。回顾5.6.3.5节可知，在执行齐次除法之后，坐标就都会位于NDC空间之中。

步骤2中从NDC空间到纹理空间变换的实现，完全取决于下列坐标变换：

$$u = 0.5x + 0.5$$
$$v = -0.5y + 0.5$$

其中，由于$x,\ y \in [-1,\ 1]$，所以$u,\ v \in [0,\ 1]$。我们为y坐标乘以–1以调转此轴的方向，这是因为NDC坐标的y轴正方向与纹理坐标的v轴正方向刚好相反。纹理空间的变换可以用矩阵的形式来表示（可参考第3章中的练习21）：

$$[x,\quad y,\quad 0,\quad 1]\begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.5 & 0.5 & 0 & 1 \end{bmatrix} = [u,\quad v,\quad 0,\quad 1]$$

我们将上述矩阵$\boldsymbol{T}$称为“纹理矩阵”，它可以将坐标由NDC空间变换到纹理空间。通过形如$\boldsymbol{VPT}$的复合变换，我们就可以将坐标从世界空间直接变换到纹理空间。将坐标乘以这个组合变换之后，还需将结果进行透视除法才能完成整个变换。至于为什么要在纹理变换之后还要执行透视除法，可参考第5章中的练习8。

20.3.1　代码实现

下列代码展示了如何生成投影纹理坐标。

```
struct VertexOut
{
  float4 PosH   : SV_POSITION;
  float3 PosW   : POSITION;
```

```
  float3 TangentW : TANGENT;
  float3 NormalW : NORMAL;
  float2 Tex    : TEXCOORD0;
  float4 ProjTex : TEXCOORD1;
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout;

   [...]

  // 把顶点变换到光源的投影空间
  vout.ProjTex = mul(float4(vin.posL, 1.0f),
    gLightWorldViewProjTexture);

   [...]

  return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
  // 通过除以w来完成投影变换
  pin.ProjTex.xyz /= pin.ProjTex.w;

  // NDC空间中的深度值
  float depth = pin.ProjTex.z;

  // 通过投影纹理坐标来采集纹理
  float4 c = gTextureMap.Sample(sampler, pin.ProjTex.xy);

  [...]
}
```

20.3.2　视锥体之外的点

在渲染流水线中，位于视锥体之外的几何体是要被裁剪掉的。但是，在我们以光源装置的视角投影几何体而为之生成投影纹理坐标时，并不必执行裁剪操作——只需简单地投影顶点即可。因此，投影机视锥

体之外的几何体会得到[0, 1]区间以外的投影纹理坐标。而处于此范围的投影纹理坐标在被采样时所享受的待遇，将与采集[0, 1]区间以外普通纹理时所用寻址模式（参见9.6节）的效果相一致。

一般来讲，我们并不希望对投影机视锥体以外的几何体进行贴图，因为这样做并没有任何意义（这些几何体并不会被投影机发出的光照射到）。常见的做法是使用颜色分量皆为0的边框颜色寻址模式。而另一种策略则是将投影机与聚光灯结合在一起，使聚光灯照射范围之外的部分不受光照（即范围之外的物体表面不会被投射光照射到）。采用聚光灯的优点是其圆锥体照射范围内中心的光照强度最大，并随着$-\mathbf{L}$与$\mathbf{d}$之间夹角的增大而平缓地减弱（其中，$\mathbf{L}$为照射到物体表面点上的光向量，$\mathbf{d}$为聚光灯的方向向量。可回顾8.12节的内容）。

20.3.3 正交投影

我们刚刚展示了如何通过透视投影（在平截头体的范围内）来进行投影纹理贴图。但是，为了投影处理过程的需要，我们应使用正交投影而非透视投影。这样一来，纹理将沿光线传播路径按正交投影方盒的z轴方向进行投影。

除了以下几点需要注意，此前讨论的投影纹理坐标相关内容都可以同样应用于正交透影。首先，在使用正交投影时，用来处理投影机视锥体范围之外点的聚光灯策略就行不通了。这是由于聚光灯的圆锥体照射范围更接近于透视投影的平截头体，但与正交投影的长方体却相差甚

远。但是，我们仍然能使用纹理寻址模式来处理位于投影机光源照射范围之外的点，这是因为正交投影仍需生成NDC坐标。若点(x, y, z)位于NDC空间中，当且仅当：

$$
\begin{aligned}
-1 \leqslant x \leqslant 1 \\
-1 \leqslant y \leqslant 1 \\
0 \leqslant z \leqslant 1
\end{aligned}
$$

其次，若采用正交投影，便无需进行除以w项计算了。也就是说，我们不必再编写下列代码：

```
// 通过除以w来完成投影操作
pin.ProjTex.xyz /= pin.ProjTex.w;
```

原因是，在完成正交投影后，坐标已经变换到了NDC空间之中。由于每个像素均免于一次除法操作，所以正交投影要比透视投影速度更快。从另一方面来讲，就算保留了除法操作也不会影响正交投影，这是因为除数是1（正交投影并不影响w坐标，因而w的值总为1）。如果在着色器代码中保留了除以w的操作，则统一使用该着色器进行透视投影与正交投影均可正常工作。当然，我们此时就需要在代码的一致性与正交投影不必要的除法操作之间进行利弊的权衡。

20.4 什么是阴影贴图

20.4.1 算法描述

阴影贴图的大致思路是，从光源的视角将场景深度以“渲染至纹理”的方式绘制到名为阴影图（shadow map）的深度缓冲区中。上述工作完成之后，我们就会得到一张从光源视角看去，由一切可见像素的深度数据所构成的阴影图（被其他像素所遮挡的像素不在此阴影图之列，因为它们在深度测试时将会失败，所以它们不是被其他像素所覆写（overwrite）就是从来没有被写入过）。

为了以光源的视角渲染场景，我们需要定义一个可以将坐标从世界空间变换到光源空间的光源观察矩阵（light view matrix），以及一个用于描述光源在世界空间中照射范围的光源投影矩阵（light projection matrix）。光源照射的体积范围可能为平截头体（透视投影）或长方体（正交投影）。通过在平截头体内嵌入聚光灯的照明圆锥体，便可用此平截头体光源体积（frustum light volume）来模拟聚光灯光源。而长方体光源体积则可用于模拟平行光光源。然而，由于平行光被限制在长方体内并且只能经过长方体体积范围，因此它只能照射到场景中的一部分区域（见图20.5）。对于能够照射到整个场景的光源（例如太阳）来说，我们就可以扩展光源体积，从而使它足以照亮全部场景。

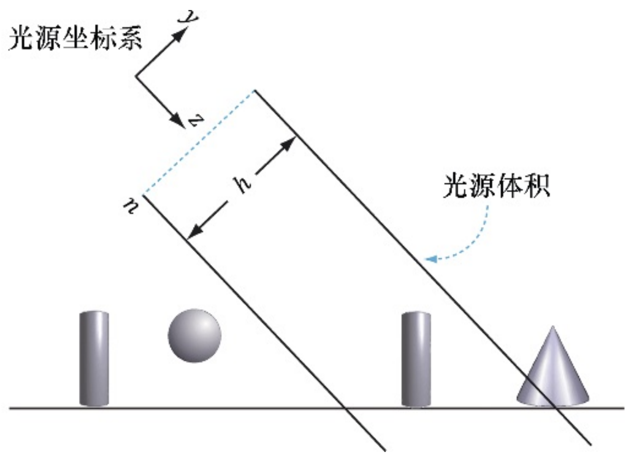


图20.5　平行光线仅在光源体积中传播，所以只有位于该体积中的场景部分才能被光照射到。如需令光源照遍整个场景，那么我们就应将光源体积设置为可容纳整个场景的大小

一旦阴影图构建完成，我们就能像往常那样以“玩家”摄像机视角来渲染场景。针对每一个要被渲染的像素p，我们都会以光源的视角来计算其深度值，并将它记为$d(p)$。另外，在使用投影纹理贴图技术时，我们还要沿着自光源至像素p的路径来对阴影图采样，以获取阴影图中所存的深度值$s(p)$。此值即为光源到像素p这一路径中，离光源最近像素的深度值。从图20.6可以发现，当且仅当$d(p) > s(p)$，像素p位于阴影范围之内（以光源的视角来看，表面上的点p被球遮挡）。因此，当且仅当$d(p) \leqslant s(p)$时，像素p位列阴影之外。

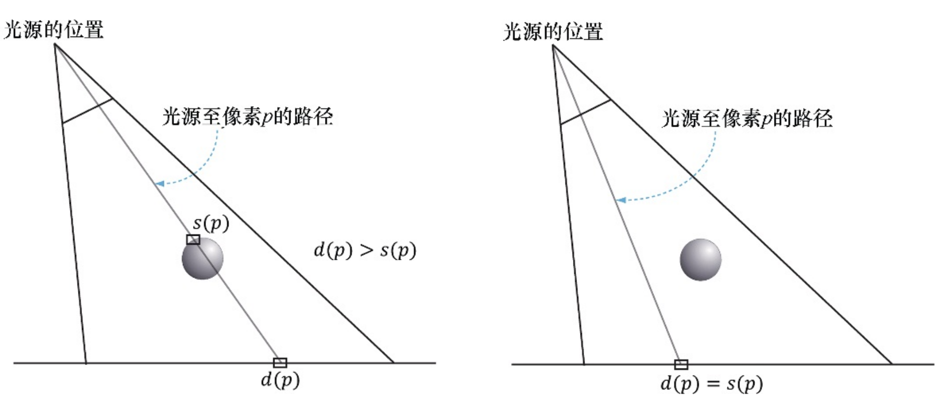


图20.6　在左图中，从光源的角度上看，像素p的深度值为$d(p)$。但是，在同一光线路径上，离

光源最近的像素深度值为$s(p)$，且$d(p) > s(p)$。因此我们便能推断出：以光源的视角来看，在像素p之前有一物体遮住了它，因此p便生活在了阴影之中。右图中，从光源角度来看，像素p的深度值为$d(p)$，而且在光源至此点这一条路径上来看，它离光源最近。即$s(p) = d(p)$，由于我们可以推理出p会感受到光明的温暖

注意 Note

深度值要用NDC坐标进行比较。这是因为阴影图其实是一种深度缓冲区，它存储着以NDC坐标表示的深度值。那么具体是怎样实现的呢？在查阅本章例程代码之后谜底自然会揭晓。

20.4.2 偏移与走样

阴影图存储的是距离光源最近的可视像素深度值，但是它的分辨率有限，以致每一个阴影图纹素都要表示场景中的一片区域。因此，阴影图只是以光源视角针对场景深度进行的离散采样，这将会导致所谓的阴影粉刺（shadow acne）等图像走样（aliasing）[1]问题（见图20.7）。

图20.8展示的简明示意图解释了为什么会发生阴影粉刺现象。一种简单的解决方案是通过恒定偏移（bias）量对阴影图的深度值进行调整。图20.9演示的就是更正该问题的过程。

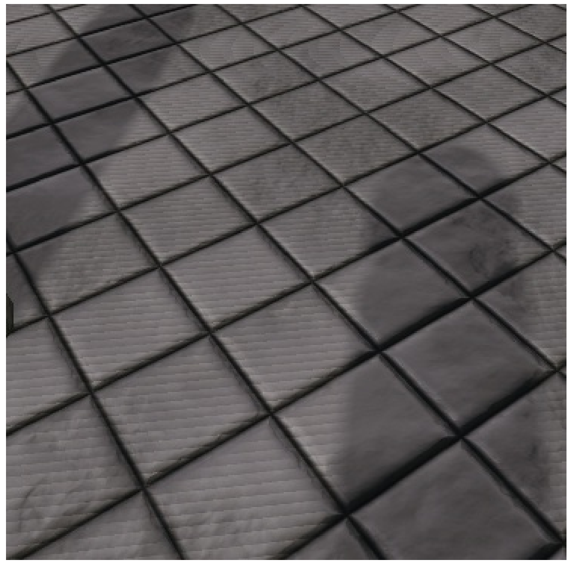


图20.7　注意图中地面上光影之间轮流交替的“阶状”条纹。这种混叠现象通常被称为阴影粉刺

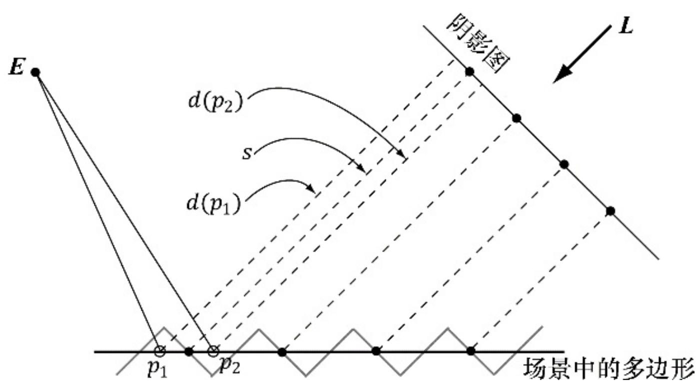


图20.8　利用阴影图采集场景中的像素深度值。从图中可以看出，由于阴影图的分辨率有限，所以每个阴影图纹素要对应于场景中的一块区域（而不是点对点的关系，一个坡面代表阴影图中一个纹素的对应范围）。从观察点$\boldsymbol{E}$查看场景中的两个点p_1与p_2，它们分别对应于两个不同的屏幕像素。但是，从光源的观察角度来看，它们却都有着相同的阴影图纹素（即$s(p_1)=s(p_2)=s$，由于分辨率的原因）。当我们在执行阴影图检测时，会得到$d(p_1)>s$以及$d(p_2)\leqslant s$这两个测试结果。这样一来，p_1将被绘制为如同它在阴影中的颜色，p_2将被渲染为好似它在阴影之外的颜色。这便会导致阴影粉刺

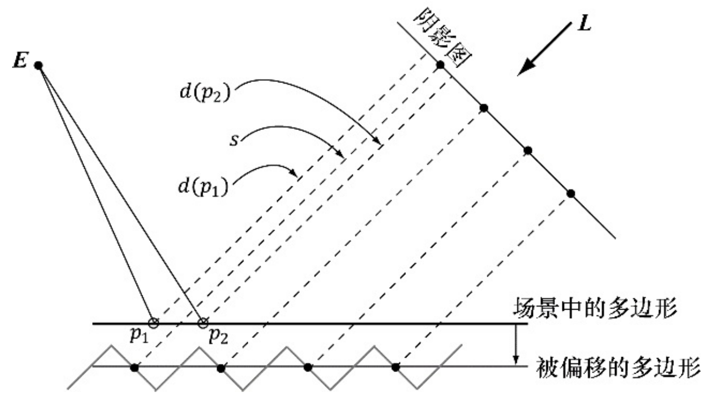


图20.9　通过偏移阴影图中的深度值来防止出现错误的阴影效果。此时，我们便可得到$d(p_1)\leqslant s$与$d(p_2)\leqslant s$。但寻找合适的深度偏移量通常要通过尝试若干

偏移量过大会导致名为**peter-panning**（彼得·潘，即小飞侠，他曾在一次逃跑时弄丢了自己的影子）的失真效果，使阴影看起来与物体相分离。

然而，并没有哪一种固定的偏移量可正确地运用于所有几何体的阴影绘制。特别是图20.11中所示的那种（从光源的角度来看）有着极大斜率的三角形，这时就需要选取更大的偏移量。但是，如果企图通过一个过大的深度偏移量来处理所有的斜边，则又会造成如图20.10所示的peter-panning问题。

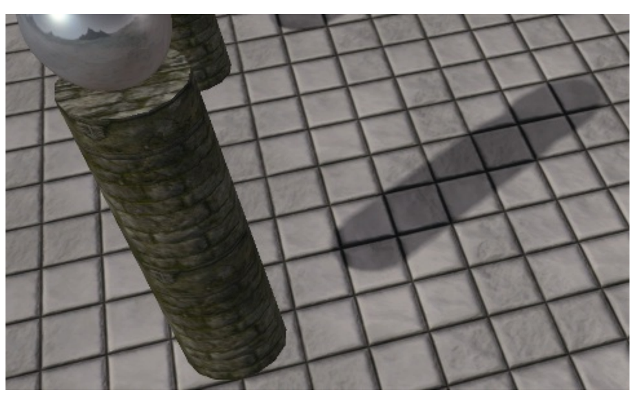


图20.10　peter-panning失真——由于过大的深度偏移量而导致的阴影与柱体分离现象

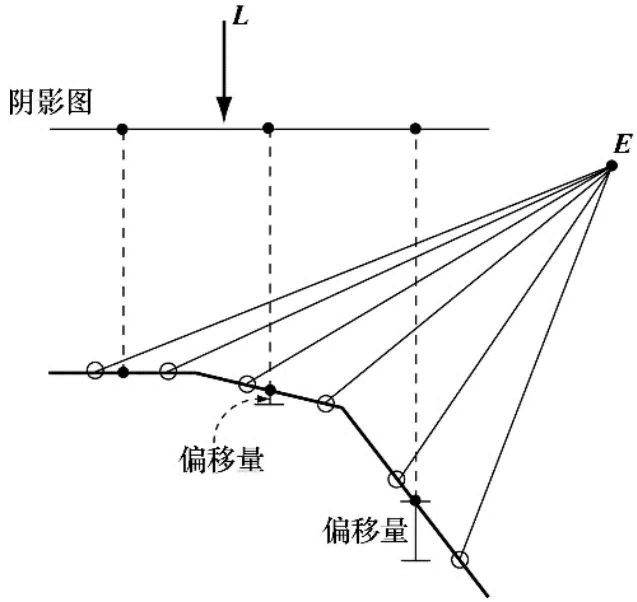


图20.11　从光源的视角来讲，有着极大斜率的多边形要比有着较小斜率的多边形采用更大的偏移量

因此，我们绘制阴影的方式就是先以光源视角度量多边形斜面的斜率，并为斜率较大的多边形应用更大的偏移量。幸运的是，图形硬件内部对此有相关技术的支持，我们通过名为斜率缩放偏移（slope-scaled-bias，也有译作斜率偏移补偿等）的光栅化状态属性就可以轻松实现。

```
typedef struct D3D12_RASTERIZER_DESC {
 [...]
 INT        DepthBias;
 FLOAT       DepthBiasClamp;
 FLOAT       SlopeScaledDepthBias;
 [...]
} D3D12_RASTERIZER_DESC;
```

1. DepthBias：一个固定的应用偏移量。对于格式为UNORM的深度缓冲区而言，可参考下列代码注释中该整数值的设置方法。

2. DepthBiasClamp：所允许的最大深度偏移量。以此来设置深度

偏移量的上限。不难想象，极其陡峭的倾斜度会致使斜率缩放偏移量过大，继而造成peter-panning失真。

3. SlopeScaledDepthBias：根据多边形的斜率来控制偏移程度的缩放因子。具体配置方法可参见下列代码注释中的公式。

注意，在将场景渲染至阴影图时，便会应用该斜率缩放偏移量。这是由于我们希望以光源的视角基于多边形的斜率而进行偏移操作，从而避免阴影失真。因此，我们就会对阴影图中的数值进行偏移计算（即由硬件将像素的深度值与偏移值相加）。在演示程序中采用的具体数值如下。

```
// [出自MSDN]
// 如果当前的深度缓冲区采用UNORM格式且与输出合并阶段绑定在一起，或深度缓冲区还没有执行绑定操作，
// 则偏移量的计算过程如下
//
// Bias = (float)DepthBias * r + SlopeScaledDepthBias * MaxDepthSlope;
//
// 这里的r是在将深度缓冲区格式转换为 float32类型后，其深度值可取到的大于0的最小可表示的值
// [MSDN援引部分结束]
//
// 对于一个24位的深度缓冲区来说，r = 1 / 2^24
//
// 例如：DepthBias = 100000 ==> 实际的DepthBias = 100000/2^24 = .006

// 这些数据极其依赖于实际场景，因此我们需要对特定场景反复尝试才能找到最合适的偏移数值
D3D12_GRAPHICS_PIPELINE_STATE_DESC smapPsoDesc = opaquePsoDesc;
smapPsoDesc.RasterizerState.DepthBias = 100000;
smapPsoDesc.RasterizerState.DepthBiasClamp = 0.0f;
smapPsoDesc.RasterizerState.SlopeScaledDepthBias = 1.0f;
```

注意 **Note**

深度偏移发生在光栅化期间（裁剪阶段之后），因此不会对几何体裁剪造成影响。

注意 **Note**

至于深度偏移的完整细节，可搜索SDK文档中的文章Depth Bias（深度偏移）。此文不仅给出了该技术相关的全部规则，而且介绍了如何使用浮点深度缓冲区进行工作。

20.4.3　百分比渐近过滤

在用投影纹理坐标(u, v)对阴影图进行采样时，往往不会命中阴影图中纹素的准确位置，而是通常位于阴影图中的4个纹素之间。若执行的是颜色贴图（color texturing），那么使用双线性插值（9.5.1节）即可解决该问题。但是[Kilgard01]却指出，我们不应对深度值采用平均值法，因为这可能导致把像素误标入阴影中这样的错误结果（出于同样的原因，我们也不能为阴影图生成mipmap）。考虑至此，我们应当对采样的结果进行插值，而不是对深度值进行插值，这种方法称为——百分比

渐近过滤（Percentage Closer Filtering，PCF）。即我们以点过滤（MIN_MAG_MIP_POINT）的方式在坐标(u, v)、$(u+\Delta x, v)$、$(u, v+\Delta x)$以及$(u+\Delta x, v+\Delta x)$处对纹理进行采样，其中$\Delta x = 1/\text{SHADOW_MAP_SIZE}$（除以的是阴影贴图的大小）。因为使用的是点采样，所以这4个采样点分别命中的是围绕坐标(u, v)最近的4个阴影图纹素s_0、s_1、s_2与s_3，如图20.12所示。接下来，我们会对这些采集的深度值进行阴影图检测，并对测试的结果开展双线性插值。

```
static const float SMAP_SIZE = 2048.0f;
static const float SMAP_DX = 1.0f / SMAP_SIZE;

...

  // 对阴影图进行采样以获取离光源最近的深度值
  float s0 = gShadowMap.Sample(gShadowSam,
          projTexC.xy).r;
  float s1 = gShadowMap.Sample(gShadowSam,
          projTexC.xy + float2(SMAP_DX, 0)).r;
  float s2 = gShadowMap.Sample(gShadowSam,
          projTexC.xy + float2(0, SMAP_DX)).r;
  float s3 = gShadowMap.Sample(gShadowSam,
          projTexC.xy + float2(SMAP_DX, SMAP_DX)).r;

  // 该像素的深度值是否小于等于阴影图中的深度值
  float result0 = depth <= s0;
  float result1 = depth <= s1;
  float result2 = depth <= s2;
  float result3 = depth <= s3;

  // 变换到纹素空间
  float2 texelPos = SMAP_SIZE*projTexC.xy;

  // 确定插值变量
  float2 t = frac( texelPos );

  // 对比较结果进行双线性插值
  return lerp( lerp(result0, result1, t.x),
         lerp(result2, result3, t.x), t.y);
```

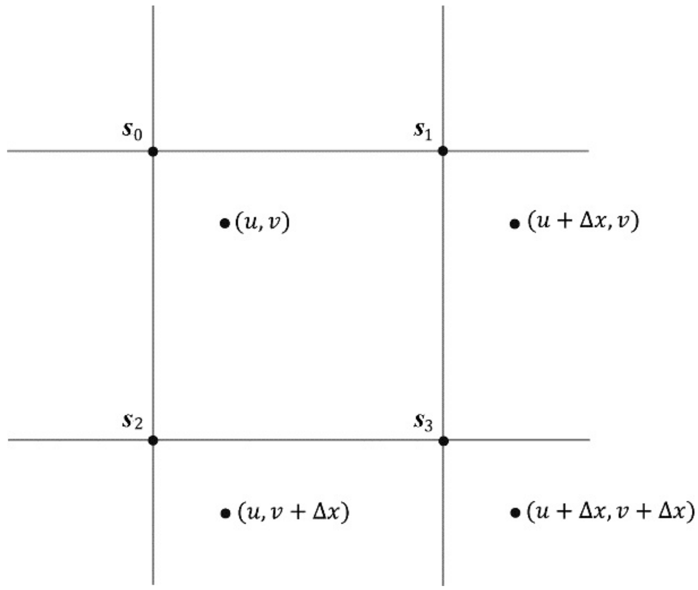


图20.12　采集4个阴影图样本

若采用这种计算方法，则一个像素就可能局部处于阴影之中，而不再是非黑即白的单选题。例如，若有4个样本，两个在阴影中，另外两个在阴影外，则该像素有50%位于阴影之中。这便使阴影内外的像素之间有了更为平滑的过渡，而非棱角分明（见图20.13）。

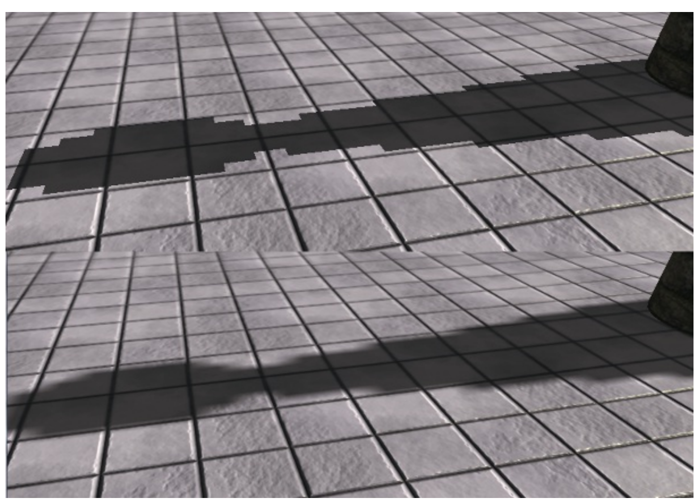


图20.13　在上侧的图中，我们可以明显看出阴影边界处的“阶梯状”锯齿走样。在下侧的图中，用过滤的方式令锯齿走样变得稍加平滑

注意 Note

HLSL中的`frac`函数返回的是输入浮点数的小数部分（即尾数，mantissa）。比如说，如果`SMAP_SIZE = 1024`且`projTex.xy = (0.23, 0.68)`，那么`texelPos = (235.52, 696.32)`且`frac(texelPos) = (0.52, 0.32)`。这些小数指示着样本之间的插值变量。而HLSL中的`lerp(x, y, s)`函数则是线性插值函数，它将返回$x + s(y - x) = (1 - s)x + sy$的计算结果。

注意 Note

尽管采用了我们自己编写的过滤方法，但阴影的效果仍然非常生硬，而且锯齿失真问题的最终处理效果还是不能令人十分满意。当然，除此之外还有许多行之有效的方法可用，如[Uralsky05]。我们也可以辅以更高分辨率的阴影图，但是需要在效果与开销之间进行取舍。

从上面的描述中也可以看出PCF过滤的主要缺点，即需要4个纹理样本。采集纹理原本就是现代GPU代价较高的操作之一，因为存储器的带宽与延迟并没有随着GPU计算能力的剧增而得到相近程度的巨大改良

[Möller08]。幸运的是，我们已经能够通过调用SampleCmpLevelZero方法来调取兼容Direct3D 11+版本的图形硬件对PCF技术的内部支持。

```
Texture2D gShadowMap : register(t1);
SamplerComparisonState gsamShadow : register(s6);

// 通过除以w来完成投影操作
shadowPosH.xyz /= shadowPosH.w;

// 在NDC空间中的深度值
float depth = shadowPosH.z;

// 自动执行4-tap PCF
gShadowMap.SampleCmpLevelZero(gsamShadow,
  shadowPosH.xy, depth).r;
```

方法名字中的LevelZero部分意味着只能在最高的mipmap层级中才能执行此函数的相应任务。这是极好的，因为我们仅希望针对阴影贴图进行采样并比较采样结果（而不会为阴影图生成mipmap链）。此方法使用的并非普通的采样器对象，而是比较采样器（comparison sampler）SamplerComparisonState。这使硬件能够执行阴影图的比较测试，且需要在过滤采样结果之前完成。对于PCF技术来说，我们需要使用D3D12_FILTER_COMPARISON_MIN_MAG_LINEAR_MIP_POINT过滤器，并将比较函数设置为LESS_EQUAL（由于对深度值进行了偏移，所以也要用到LESS比较函数）。此方法的前两个参数分别为比较采样器对象以及纹理坐标，第三个参数则是与阴影图样本相比较的数值。在将比较数值设置为depth并把比较函数设定为LESS_EQUAL之后，它将执行以下比较操作：

```
float result0 = depth <= s0;
float result1 = depth <= s1;
float result2 = depth <= s2;
```

```
float result3 = depth <= s3;
```

接着，在通过硬件线性插值得到最终的计算结果之后，便完成了PCF的整个工作流程。

下列代码展示了如何为阴影贴图配置比较采样器。

```
const CD3DX12_STATIC_SAMPLER_DESC shadow(
  6, // 着色器寄存器(shaderRegister)
  D3D12_FILTER_COMPARISON_MIN_MAG_LINEAR_MIP_POINT, // 过滤器类型(filter)
  D3D12_TEXTURE_ADDRESS_MODE_BORDER, // U轴所用的寻址模式(addressU)
  D3D12_TEXTURE_ADDRESS_MODE_BORDER, // V轴所用的寻址模式(addressV)
  D3D12_TEXTURE_ADDRESS_MODE_BORDER, // W轴所用的寻址模式(addressW)
  0.0f,                 // mipmap层级偏移量(mipLODBias)
  16,                   // 最大各向异性值(maxAnisotropy)
  D3D12_COMPARISON_FUNC_LESS_EQUAL,
  D3D12_STATIC_BORDER_COLOR_OPAQUE_BLACK);
```

注意 Note

根据SDK文档所述，只有R32_FLOAT_X8X24_TYPELESS格式、R32_FLOAT，R24_UNORM_X8_TYPELESS格式与R16_UNORM格式才可应用于比较过滤器。

到目前为止，我们在本节中一直使用的都是4-tap PCF核（4-tap PCF kernel，即输入4个样本来执行的PCF）。PCF核越大，阴影的边缘轮廓也就越丰满、越平滑，当然，花费在调用SampleCmpLevelZero函数上的开销也就越大。在演示程序中，我们是按3×3正方形的过滤模式来执

行PCF。由于每次调用`SampleCmpLevelZero`函数实际所执行的都是4-tap PCF，所以在进行上述PCF的过程中，共需要使用阴影图中的4×4个独立采样点（根据我们所用的3×3模式可知，过滤的过程中会重复使用部分采样点）。采用过大的过滤核会导致之前所述的阴影粉刺问题，原因以及解决方案我们将在20.5节中展开讨论。

一个显而易见的事实是，PCF技术只需在阴影的边缘执行，因为阴影内外两部分并不涉及混合操作（意即阴影内外非黑即白，只有边缘才是渐变的）。基于此，也就只要能对阴影边缘的PCF设计相应的处理方案就好了。[Isidoro06b]指出了这样一种运用着色器代码中的动态分支技术：“如果要处理的部分是阴影边缘就采用代价高昂的PCF技术，否则就仅对一幅阴影图进行采样。”

需要注意的是，如果我们所用的PCF核足够大（即5×5采样点及其以上规模），那么按上述方法做才合算（因为动态分支也有开销）。但是，按一般的建议来讲，我们还是应当根据具体需求在开销和效果之间做出权衡。

注意，实际工程中所用的PCF核未必一定是方形的过滤栅格。不少文献也已指出，随机拾取点（randomly picking point）也可作为PCF核。

20.4.4 构建阴影图

实现阴影贴图的第一步就是构建阴影图。为此，首先需要创建一个`ShadowMap`实例。

```
mShadowMap = std::make_unique<ShadowMap>(
  md3dDevice.Get(), 2048, 2048);
```

接着，定义一个光源观察矩阵以及一个投影矩阵（用来表示光源坐标系与视景体）。光源观察矩阵要以主光源的视角构建，而光源视景体要根据整个场景的包围球来进行计算。

```
DirectX::BoundingSphere mSceneBounds;

ShadowMapApp::ShadowMapApp(HINSTANCE hInstance)
  : D3DApp(hInstance)
{
  // 由于我们知道当前场景是如何构建出来的，因此可手动估算场景的包围球
  // 中心位于世界空间的原点，且宽度为20，深度为30.0f的栅格是场景中“最宽的物体”。
而在实际的项
  // 目中，我们若要计算此包围球，通常就需要遍历世界空间中的每个顶点的位置
  mSceneBounds.Center = XMFLOAT3(0.0f, 0.0f, 0.0f);
  mSceneBounds.Radius = sqrtf(10.0f*10.0f + 15.0f*15.0f);
}

void ShadowMapApp::Update(const GameTimer& gt)
{
  [...]

  //
  // 根据时间的流逝动态调整各光源（以及物体阴影）
  //

  mLightRotationAngle += 0.1f*gt.DeltaTime();

  XMMATRIX R = XMMatrixRotationY(mLightRotationAngle);
  for(int i = 0; i < 3; ++i)
  {
    XMVECTOR lightDir = XMLoadFloat3(&mBaseLightDirections[i]);
    lightDir = XMVector3TransformNormal(lightDir, R);
    XMStoreFloat3(&mRotatedLightDirections[i], lightDir);
  }

  AnimateMaterials(gt);
  UpdateObjectCBs(gt);
  UpdateMaterialBuffer(gt);
  UpdateShadowTransform(gt);
  UpdateMainPassCB(gt);
```

```
  UpdateShadowPassCB(gt);
}

void ShadowMapApp::UpdateShadowTransform(const GameTimer& gt)
{
  // 只有第一个“主”光源才投射出物体的阴影
  XMVECTOR lightDir = XMLoadFloat3(&mRotatedLightDirections[0]);
  XMVECTOR lightPos = -2.0f*mSceneBounds.Radius*lightDir;
  XMVECTOR targetPos = XMLoadFloat3(&mSceneBounds.Center);
  XMVECTOR lightUp = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);
  XMMATRIX lightView = XMMatrixLookAtLH(lightPos, targetPos, lightUp);

  XMStoreFloat3(&mLightPosW, lightPos);

  // 将包围球变换到光源空间
  XMFLOAT3 sphereCenterLS;
  XMStoreFloat3(&sphereCenterLS, XMVector3TransformCoord(targetPos,
    lightView));

  // 位于光源空间中包围场景的正交投影视景体
  float l = sphereCenterLS.x - mSceneBounds.Radius;
  float b = sphereCenterLS.y - mSceneBounds.Radius;
  float n = sphereCenterLS.z - mSceneBounds.Radius;
  float r = sphereCenterLS.x + mSceneBounds.Radius;
  float t = sphereCenterLS.y + mSceneBounds.Radius;
  float f = sphereCenterLS.z + mSceneBounds.Radius;

  mLightNearZ = n;
  mLightFarZ = f;
  XMMATRIX lightProj = XMMatrixOrthographicOffCenterLH(l, r, b, t, n, f);

  // 将坐标从范围为[-1,+1]^2的NDC空间变换到范围为[0,1]^2的纹理空间
  XMMATRIX T(
    0.5f, 0.0f, 0.0f, 0.0f,
    0.0f, -0.5f, 0.0f, 0.0f,
    0.0f, 0.0f, 1.0f, 0.0f,
    0.5f, 0.5f, 0.0f, 1.0f);

  XMMATRIX S = lightView*lightProj*T;
  XMStoreFloat4x4(&mLightView, lightView);
  XMStoreFloat4x4(&mLightProj, lightProj);
  XMStoreFloat4x4(&mShadowTransform, S);
}
```

将场景渲染至阴影图要这样实现：

```
void ShadowMapApp::DrawSceneToShadowMap()
{
  mCommandList->RSSetViewports(1, &mShadowMap->Viewport());
  mCommandList->RSSetScissorRects(1, &mShadowMap->ScissorRect());

  // 将资源状态改变为DEPTH_WRITE
  mCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
    mShadowMap->Resource(),
    D3D12_RESOURCE_STATE_GENERIC_READ,
    D3D12_RESOURCE_STATE_DEPTH_WRITE));

  UINT passCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof
   (PassConstants));

  // 清理后台缓冲区以及深度缓冲区
  mCommandList->ClearDepthStencilView(mShadowMap->Dsv(),
    D3D12_CLEAR_FLAG_DEPTH | D3D12_CLEAR_FLAG_STENCIL, 1.0f, 0, 0, nullptr
);

  // 由于仅向深度缓冲区绘制数据，因此将渲染目标设为空。这样一来将禁止颜色数据向渲
染目标的写操作。
  // 注意，此时也一定要把可用（处于启用状态）PSO中的渲染目标数量指定为0
  mCommandList->OMSetRenderTargets(0, nullptr, false, &mShadowMap->Dsv());

  // 为阴影图渲染过程绑定所需的常量缓冲区
  auto passCB = mCurrFrameResource->PassCB->Resource();
  D3D12_GPU_VIRTUAL_ADDRESS passCBAddress = passCB->GetGPUVirtualAddress()
    + 1*passCBByteSize;
  mCommandList->SetGraphicsRootConstantBufferView(1, passCBAddress);

  mCommandList->SetPipelineState(mPSOs["shadow_opaque"].Get());

  DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Opaque
]);

  // 将资源状态改变回GENERIC_READ，使我们能够从着色器中读取此纹理
  mCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(
    mShadowMap->Resource(),
    D3D12_RESOURCE_STATE_DEPTH_WRITE,
    D3D12_RESOURCE_STATE_GENERIC_READ));
}
```

可以看出，我们在这里设置了一个空的渲染目标，其实就是禁止了颜色数据的写操作。这样做的原因是在将场景渲染至阴影图时，我们只

关心相对于光源的场景深度值。显卡针对仅绘制深度数据的情况进行了优化，因此仅绘制深度值的渲染过程明显要快于同时绘制颜色数据与深度值的渲染过程。此时，我们也必须将处于活动状态的流水线状态对象的渲染目标个数指定为0。

```
D3D12_GRAPHICS_PIPELINE_STATE_DESC smapPsoDesc = opaquePsoDesc;
smapPsoDesc.RasterizerState.DepthBias = 100000;
smapPsoDesc.RasterizerState.DepthBiasClamp = 0.0f;
smapPsoDesc.RasterizerState.SlopeScaledDepthBias = 1.0f;
smapPsoDesc.pRootSignature = mRootSignature.Get();
smapPsoDesc.VS =
{
  reinterpret_cast(mShaders["shadowVS"]->GetBufferPointer()),
  mShaders["shadowVS"]->GetBufferSize()
};
smapPsoDesc.PS =
{
  reinterpret_cast(mShaders["shadowOpaquePS"]->GetBufferPointer()),
  mShaders["shadowOpaquePS"]->GetBufferSize()
};
// 阴影图的渲染过程无须涉及渲染目标

smapPsoDesc.RTVFormats[0] = DXGI_FORMAT_UNKNOWN;
smapPsoDesc.NumRenderTargets = 0;
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(
  &smapPsoDesc, IID_PPV_ARGS(&mPSOs["shadow_opaque"])));
```

以光源的视角渲染场景的着色器程序十分简单，这是因为只需构建阴影图即可，所以也就用不到那些复杂的代码。

```
//*********************************************************************
// Shadows.hlsl的作者是Frank Luna (C) 2015 版权所有
//*********************************************************************

// 包含公用的HLSL代码
#include "Common.hlsl"

struct VertexIn
{
  float3 PosL  : POSITION;
```

```
  float2 TexC  : TEXCOORD;
};

struct VertexOut
{
  float4 PosH  : SV_POSITION;
  float2 TexC  : TEXCOORD;
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout = (VertexOut)0.0f;

  MaterialData matData = gMaterialData[gMaterialIndex];

  // 将顶点变换到世界空间
  float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);

  // 将顶点变换至齐次裁剪空间
  vout.PosH = mul(posW, gViewProj);

  // 为三角形插值而输出顶点属性
  float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
  vout.TexC = mul(texC, matData.MatTransform).xy;

  return vout;
}

// 这段代码仅用于需要进行alpha裁剪的几何图形，以此使阴影正确地显现出来
// 如果待处理的几何图形无须执行此操作，则可以在深度渲染过程中使用无内容（null）的
像素着色器
void PS(VertexOut pin)
{
  // 获取材质数据
  MaterialData matData = gMaterialData[gMaterialIndex];
  float4 diffuseAlbedo = matData.DiffuseAlbedo;
  uint diffuseMapIndex = matData.DiffuseMapIndex;

  // 在数组中动态地查找纹理
  diffuseAlbedo *= gTextureMaps[diffuseMapIndex].
    Sample(gsamAnisotropicWrap, pin.TexC);

#ifdef ALPHA_TEST
  // 如果像素的alpha值 < 0.1,丢弃该像素。我们应在着色器中尽早执行这项测试，以使
满足条件的
  // 像素尽早地退出着色器，以此来跳过没必要执行的后续代码
  clip(diffuseAlbedo.a - 0.1f);
#endif
```

```
}
```

可以看出像素着色器并没有返回任何数据，这是因为在深度绘制过程中只需输出深度值即可。其间，像素着色器仅用于裁剪具有0或较小alpha值的像素片段，我们假设这种像素片段都是完全透明的。以图20.14中的树叶纹理为例。在此，我们只希望将具有白色alpha值的像素绘制到阴影图中。为了实现此目标，我们将其划分为两种情况：一种是需要执行alpha裁剪（alpha clip）操作的，另一种则不需要此操作。如果待处理的几何图形不需要执行alpha裁剪，那么我们就可以给它绑定一个空的像素着色器，这样一来，其执行速度将快于绑定上述像素着色器（此着色器仅对一种纹理进行采样还要进行裁剪操作的）的情况。



图20.14　叶片纹理

注意 Note

考虑到篇幅问题，我们仅对经镶嵌化处理的几何体渲染深度数据的着色器做简要分析。在将镶嵌化处理后的几何体绘制到阴影图的时候，

我们也需要按同样的镶嵌方式对该几何体进行镶嵌化处理，再将其绘制到后台缓冲区（即根据玩家的观察点到该几何体的距离对它进行绘制）。这样做的原因是为了保持一致性，即从观察者视角查看的几何体应当与光源视角观察的几何体相同。话虽如此，但如果镶嵌化几何体的形状变化不太大，那么这些改变也不太能在阴影中得到体现。因此，一种可能的优化方式是，在渲染几何体的阴影图时，不必对几何体进行镶嵌化处理。此优化要考虑精度与速度的取舍。

20.4.5　阴影因子

阴影因子（shadow factor）是我们为光照方程新添加的一种范围在[0, 1]的标量系数。其值为0，表示位于阴影中的点；值为1，则代表此点在阴影之外。在进行PCF（20.4.3节）时，一个点可能会部分处于阴影之中，在这种情况下，阴影因子将位于0～1。`CalcShadowFactor`（计算阴影因子）函数的实现位于Common.hlsl文件之中。

```
float CalcShadowFactor(float4 shadowPosH)
{
  // 通过除以w来实现投影变换
  shadowPosH.xyz /= shadowPosH.w;

  // NDC空间中的深度值
  float depth = shadowPosH.z;

  uint width, height, numMips;
  gShadowMap.GetDimensions(0, width, height, numMips);

  // 纹素的大小
  float dx = 1.0f / (float)width;
```

```
  float percentLit = 0.0f;
  const float2 offsets[9] =
  {
    float2(-dx, -dx), float2(0.0f, -dx), float2(dx, -dx),
    float2(-dx, 0.0f), float2(0.0f, 0.0f), float2(dx, 0.0f),
    float2(-dx, +dx), float2(0.0f, +dx), float2(dx, +dx)
  };

  [unroll]
  for(int i = 0; i < 9; ++i)
  {
    percentLit += gShadowMap.SampleCmpLevelZero(gsamShadow,
      shadowPosH.xy + offsets[i], depth).r;
  }

  return percentLit / 9.0f;
}
```

在我们所用的模型中，阴影因子将与直接光照（漫反射光与镜面反射光）项相乘。

```
// 只有第一个光源才投射物体阴影
float3 shadowFactor = float3(1.0f, 1.0f, 1.0f);
shadowFactor[0] = CalcShadowFactor(pin.ShadowPosH);

const float shininess = (1.0f - roughness) * normalMapSample.a;
Material mat = { diffuseAlbedo, fresnelR0, shininess };
float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
  bumpedNormalW, toEyeW, shadowFactor);

float4 ComputeLighting(Light gLights[MaxLights], Material mat,
            float3 pos, float3 normal, float3 toEye,
            float3 shadowFactor)
{
  float3 result = 0.0f;

  int i = 0;

#if (NUM_DIR_LIGHTS > 0)
  for(i = 0; i < NUM_DIR_LIGHTS; ++i)
  {
    result += shadowFactor[i] * ComputeDirectionalLight(gLights[i],
    mat, normal, toEye);
  }
```

```
#endif

#if (NUM_POINT_LIGHTS > 0)
  for(i = NUM_DIR_LIGHTS; i < NUM_DIR_LIGHTS+NUM_POINT_LIGHTS; ++i)
  {
    result += ComputePointLight(gLights[i], mat, pos, normal, toEye);
  }
#endif

#if (NUM_SPOT_LIGHTS > 0)
  for(i = NUM_DIR_LIGHTS + NUM_POINT_LIGHTS; i < NUM_DIR_LIGHTS +
    NUM_POINT_LIGHTS + NUM_SPOT_LIGHTS; ++i)
  {
    result += ComputeSpotLight(gLights[i], mat, pos, normal, toEye);
  }
#endif

  return float4(result, 0.0f);
}
```

由于环境光是间接光，所以阴影因子不会对它产生影响。而且，阴影因子也不会对来自环境图（environment map）的反射光构成影响。

20.4.6 阴影图检测

通过以光源的视角渲染场景来构建阴影图之后，我们就可以在主渲染过程中对阴影图进行采样，以确定某像素是否位于阴影之内。问题的关键在于为每个像素p都要计算$d(p)$与$s(p)$。把点变换到光源的NDC空间便可以求出其对应的$d(p)$值，此时，该点的z坐标即为此点位于光源空间中的归一化深度值。在光源的视景体中运用投影纹理贴图技术，投影出场景的阴影图就可以求出值$s(p)$。注意，在进行这一系列操作之后，$d(p)$与$s(p)$都将被表示在光源的NDC空间之中，因此它们就可以直接展开比较。再利用变换矩阵`gShadowTransform`便可以将坐标由世界空间变换

到阴影图纹理空间（参见20.3节）。

```
// 在顶点着色器中，为场景阴影图而生成的投影纹理坐标
vout.ShadowPosH = mul(posW, gShadowTransform);

// 在像素着色器中执行阴影图检测
float3 shadowFactor = float3(1.0f, 1.0f, 1.0f);
shadowFactor[0] = CalcShadowFactor(pin.ShadowPosH);
```

最后，别忘了将`gShadowTransform`矩阵存于一个渲染过程常量之中。

20.4.7　渲染阴影图

在本章的演示程序中，我们也将阴影图渲染在了屏幕右下角的四边形之中。从该图中，我们可以观察到阴影图在每一帧中的变化。前面曾讲过，阴影图其实是一种深度缓冲区纹理，我们可以为它创建一个SRV（着色器资源视图），以便在着色器程序中对它进行采样。由于阴影图中每个像素存储的都是一个一维数据（即一个深度值），所以我们将它渲染为一幅灰度图像。图20.15所示的是一张“Shadow Map”（阴影图）演示程序的效果。

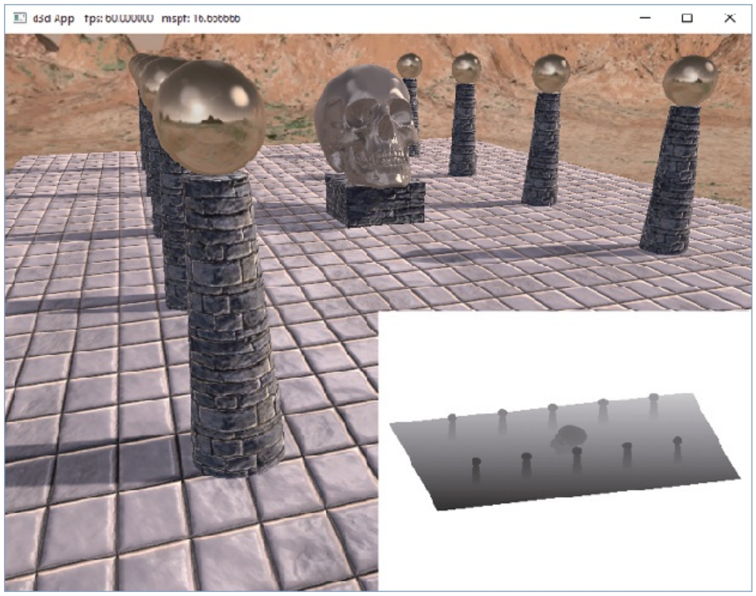


图20.15　阴影图例程的效果

20.5 过大的PCF核

在本节中，我们将讨论使用过大的PCF核所引发的问题。我们在演示程序中所用的PCF核不是很大，所以这一节在某种意义上来说是选学内容，但是在此会接触到一些有趣的套路。

图20.16所示的是我们为观察点可见像素p执行阴影检测的计算过程。如果没有使用PCF技术，就计算距离$d = d(p)$，并将它与相应的阴影图数据$s_0 = s(p)$进行比较；若采用PCF技术，我们还应令d与附近的纹理图数据s_{-1}与s_1进行比较。但是，使d分别与s_{-1}和s_1进行比较其实是不合理的，这是因为纹素s_{-1}与s_1所描述深度值的场景区域有可能与p所在的多边形并不相同。

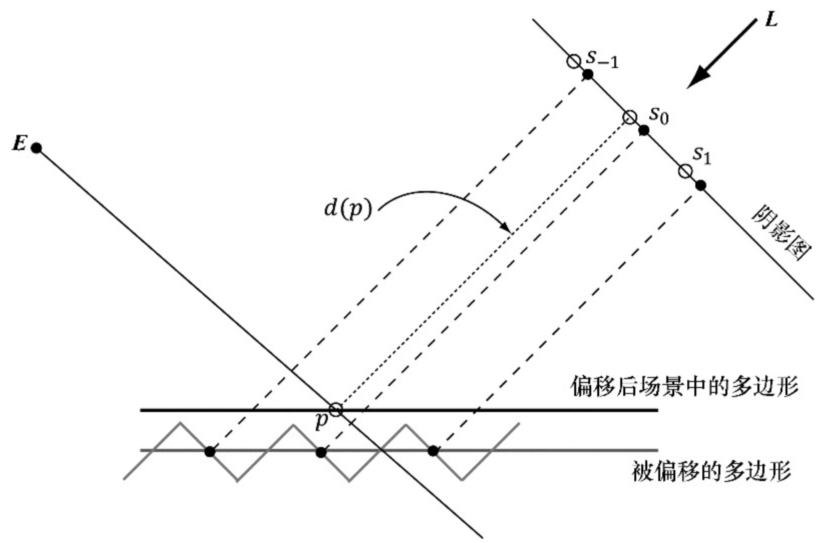


图20.16　令深度值$d(p)$与s_0进行比较是正确的，因为纹素s_0所覆盖的场景区域必然包括像素p。然而，使$d(p)$分别与s_{-1}和s_1进行比较却是错误的，这是因为s_{-1}与s_1覆盖的场景区域与像素p无关

图20.16所示的情景就是一宗由PCF引起的误判。特别是在我们进行以下阴影图检测的时候：

$$lit_0 = d \leqslant s_0\text{（真）}$$

$$lit_{-1} = d \leqslant s_{-1}\text{（真）}$$

$$lit_1 = d \leqslant s_1\text{（假）}$$

如果按这个检测结果进行插值，我们将得到点p的1/3位于阴影中这一错误的结论，而实际上点p根本没有被任何东西遮挡住。

观察图20.16可知，若将深度偏移量增大便可修正这个问题。然而，在这个示例中，我们只能对阴影图中邻近的纹素进行采样。如果我们继续扩大PCF核，那么很可能还需要继续增大偏移量。总的来讲，针对小的PCF核来说，只需像20.4.2节中介绍的那样，简单地进行深度偏移便可以无后顾之忧地解决该问题。而对于像5×5或9×9这样规模较大的PCF核来说，虽然确实可以生成过渡更加自然的软阴影（soft shadow，柔和阴影），但同时也可能引起一些严重的问题。

20.5.1 ddx函数与ddy函数

在研究较大PCF核问题的解决方案之前，首先需要讨论ddx与ddy这两种HLSL函数。它们分别近似地计算出$\partial \boldsymbol{p}/\partial x$与$\partial \boldsymbol{p}/\partial y$的值，其中，$x$是指屏幕空间的$x$轴，$y$则为屏幕空间的$y$轴。有了这两种函数，我们便可以确定相邻像素之间某属性值$\boldsymbol{p}$的变化量。例如，这两个导函数可以用

于：

1．估算相邻像素的颜色变化量。

2．估算相邻像素的深度变化量。

3．估算相邻像素的法线变化量。

硬件估算这些偏导数的过程并不复杂。它会按2×2像素规模的四边形进行并行处理，以前向差分方程$q_{x+1,y} - q_{x,y}$来估算x方向上的偏导数（即估算由像素(x, y)到像素$(x+1, y)$的变化量$\boldsymbol{q}$），并以类似的方法来计算y方向上的偏导数。

20.5.2　较大PCF核问题的解决方案

本节所描述的方案来自于[Tuft10]。该策略需要满足的前提是假设p与其相邻像素位于同一平面之内。实际应用中未必处处满足此条件，但是若依此方案解决问题，最好还是如前提条件所述。

设$\boldsymbol{p} = (u, v, z)$为光源空间中的坐标，坐标$(u, v)$用于索引阴影图，$z$值则表示阴影图检测过程中所用的光源到该点的距离。我们可以用ddx与ddy这两个函数来分别计算位于多边形所在切平面内的向量$\frac{\partial \boldsymbol{p}}{\partial x} = \left(\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x}, \frac{\partial z}{\partial x}\right)$与向量$\frac{\partial \boldsymbol{p}}{\partial y} = \left(\frac{\partial u}{\partial y}, \frac{\partial v}{\partial y}, \frac{\partial z}{\partial y}\right)$。此两个向量反映了在屏幕空间与推算光源空间里移动中的单位对应关系。特别地，如果我们在屏幕空间中移动了$(\Delta x, \Delta y)$个单位，那么在光源空间中应按切向量方向对

应地移动$\Delta x\left(\frac{\partial u}{\partial x},\frac{\partial v}{\partial x},\frac{\partial z}{\partial x}\right)+\Delta y\left(\frac{\partial u}{\partial y},\frac{\partial v}{\partial y},\frac{\partial z}{\partial y}\right)$个单位。在此，我们先暂时忽略深度项，如果在屏幕空间中移动了$(\Delta x,\Delta y)$个单位，则应在光源空间的$\boldsymbol{uv}$平面内对应地移动$\Delta x\left(\frac{\partial u}{\partial x},\frac{\partial v}{\partial x}\right)+\Delta y\left(\frac{\partial u}{\partial y},\frac{\partial v}{\partial y}\right)$个单位。这个操作可由下列矩阵方程来表示：

$$[\Delta x,\Delta y]\begin{bmatrix}\frac{\partial u}{\partial x} & \frac{\partial v}{\partial x}\\ \frac{\partial u}{\partial y} & \frac{\partial v}{\partial y}\end{bmatrix}=\Delta x\left(\frac{\partial u}{\partial x},\frac{\partial v}{\partial x}\right)+\Delta y\left(\frac{\partial u}{\partial y},\frac{\partial v}{\partial y}\right)=[\Delta u,\Delta v]$$

因此

$$\begin{aligned}[\Delta x,\Delta y]&=[\Delta u,\Delta v]\begin{bmatrix}\frac{\partial u}{\partial x} & \frac{\partial v}{\partial x}\\ \frac{\partial u}{\partial y} & \frac{\partial v}{\partial y}\end{bmatrix}^{-1}\\ &=[\Delta u,\Delta v]\frac{1}{\frac{\partial u}{\partial x}\frac{\partial v}{\partial y}-\frac{\partial v}{\partial x}\frac{\partial u}{\partial y}}\begin{bmatrix}\frac{\partial v}{\partial y} & -\frac{\partial v}{\partial x}\\ -\frac{\partial u}{\partial y} & \frac{\partial u}{\partial x}\end{bmatrix}\end{aligned}\qquad（20.1）$$

注意 Note

回顾第2章介绍的内容可知：

$$\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}^{-1}=\frac{1}{A_{11}A_{22}-A_{12}A_{21}}\begin{bmatrix}A_{22} & -A_{12}\\ -A_{21} & A_{11}\end{bmatrix}$$

这个新推导的方程反映出，如果在光源空间中的$\boldsymbol{uv}$平面内移动了$(\Delta u,\Delta v)$个单位，那么在平面空间中则移动了$(\Delta x,\Delta y)$个单位。我们为什

么要在式（20.1）上花费这么多时间呢？这是因为在构建PCF核时，我们需要偏移纹理坐标来采集阴影图中目标点的邻近数据。

```
// 纹素的大小
    float dx = 1.0f / (float)width;

    float percentLit = 0.0f;
    const float2 offsets[9] =
    {
        float2(-dx, -dx), float2(0.0f, -dx), float2(dx, -dx),
        float2(-dx, 0.0f), float2(0.0f, 0.0f), float2(dx, 0.0f),
        float2(-dx, +dx), float2(0.0f, +dx), float2(dx, +dx)
    };

    // 3x3的方形过滤器模式。每个样本要执行一次4-tap PCF操作
     [unroll]
    for(int i = 0; i < 9; ++i)
    {
        percentLit += gShadowMap.SampleCmpLevelZero(gsamShadow,
                shadowPosH.xy + offsets[i], depth).r;
    }
```

换句话说，我们就相当于知道了（对应于屏幕空间的）光源空间中$\boldsymbol{uv}$平面内的偏移量——$(\Delta u, \Delta v)$。从式（20.1）可以看出：如果在光源空间中移动了$(\Delta u, \Delta v)$个单位，那么在屏幕空间中移动的单位势必为$(\Delta x, \Delta y)$。

现在重新来考虑前面忽略的深度项。如果我们在屏幕空间中移动了$(\Delta x, \Delta y)$个单位，那么就会在光源空间中的深度方向移动$\Delta z = \Delta x \frac{\partial z}{\partial x} + \Delta y \frac{\partial z}{\partial y}$。这样一来，当偏移纹理坐标来执行PCF操作时，我们就该相应地修改深度值$z' = z + \Delta z$来进行深度检测（见图20.17）。

下面我们来总结一下较大PCF核的处理思路：

1．在实现PCF的过程中，我们会通过偏移纹理坐标来采集阴影图中目标点的邻近的数据。因此，对于每个样本来说，我们知道其偏移量$(\Delta u, \Delta v)$。

2．借助式（20.1）便可以求出在光源空间中偏移$(\Delta u, \Delta v)$个单位时，屏幕空间中的相应偏移量$(\Delta x, \Delta y)$。

3．解得$(\Delta x, \Delta y)$之后，我们就能运用$\Delta z = \Delta x \frac{\partial z}{\partial x} + \Delta y \frac{\partial z}{\partial y}$来求出光源空间中的深度变化量。

SDirectX 11 SDK 中的“CascadedShadowMaps11”演示程序就是以`CalculateRightAndUpTexel- DepthDeltas`函数与`CalculatePCFPercentLit`函数实现了上述方法。

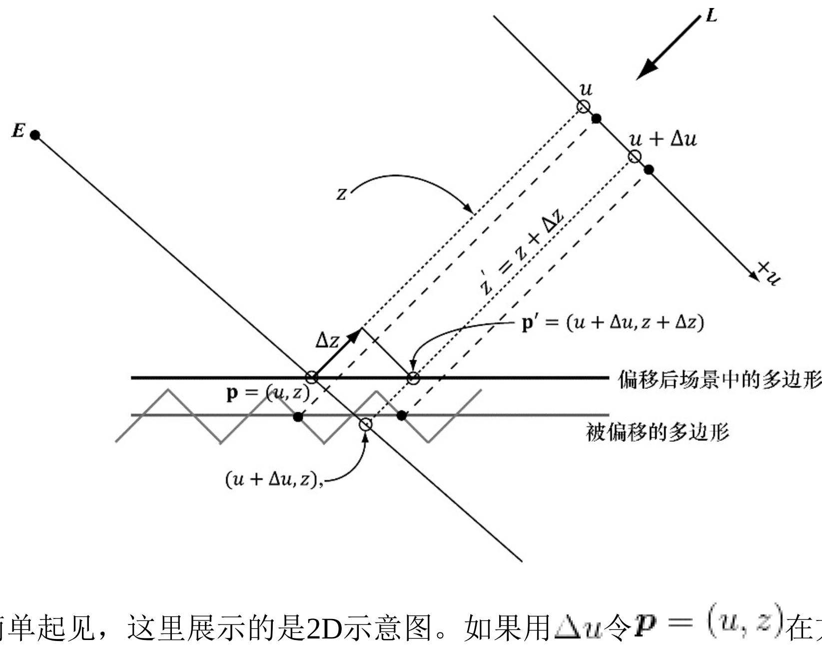


图20.17　为简单起见，这里展示的是2D示意图。如果用Δu令$\boldsymbol{p} = (u, z)$在方向u上进行偏移，并得到$(u + \Delta u, z)$，那么，我们还需要以Δz对z坐标进行偏移，使该点仍处于多边形之上，这将得到$\boldsymbol{p}' = (u + \Delta u, z + \Delta z)$

20.5.3　较大PCF核问题的另一种解决方案

本节介绍的方案出自于[Isidoro06]，这是一种与前一节中所述的方法稍有不同的改良方案的计算方法。

设$\boldsymbol{p}=(u,v,z)$为光源空间中的坐标。坐标(u,v)用于对阴影图进行索引，z值则表示阴影图检测过程中光源至该点的距离。我们可以用函数ddx与ddy计算$\frac{\partial \boldsymbol{p}}{\partial x}=\left(\frac{\partial u}{\partial x},\frac{\partial v}{\partial x},\frac{\partial z}{\partial x}\right)$与$\frac{\partial \boldsymbol{p}}{\partial y}=\left(\frac{\partial u}{\partial y},\frac{\partial v}{\partial y},\frac{\partial z}{\partial y}\right)$。

事实上，我们可以把式中的偏导数表示为$u=u(x,y)$、$v=v(x,y)$与$z=z(x,y)$这些自变量为x、y的函数。而且，也可以把z看作是自变量为u、v的函数，即$z=z(u,v)$。这是因为我们在光源空间中是按u与v的方向进行移动，所以深度z是在沿着多边形所在平面移动的过程中而产生变化的。根据链式法则（chain rule），有：

$$\frac{\partial z}{\partial x}=\frac{\partial z}{\partial u}\frac{\partial u}{\partial x}+\frac{\partial z}{\partial v}\frac{\partial v}{\partial x}$$
$$\frac{\partial z}{\partial y}=\frac{\partial z}{\partial u}\frac{\partial u}{\partial y}+\frac{\partial z}{\partial v}\frac{\partial v}{\partial y}$$

或者用矩阵表示法记作：

$$\begin{bmatrix}\frac{\partial z}{\partial x} & \frac{\partial z}{\partial y}\end{bmatrix}=\begin{bmatrix}\frac{\partial z}{\partial u} & \frac{\partial z}{\partial v}\end{bmatrix}\begin{bmatrix}\frac{\partial u}{\partial x} & \frac{\partial u}{\partial y}\\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y}\end{bmatrix}$$

根据矩阵的逆，可将公式变换为：

$$\begin{bmatrix}\frac{\partial z}{\partial u} & \frac{\partial z}{\partial v}\end{bmatrix} = \begin{bmatrix}\frac{\partial z}{\partial x} & \frac{\partial z}{\partial y}\end{bmatrix} \begin{bmatrix}\frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y}\end{bmatrix}^{-1}$$
$$= \frac{\begin{bmatrix}\frac{\partial z}{\partial x} & \frac{\partial z}{\partial y}\end{bmatrix}}{\frac{\partial u}{\partial x}\frac{\partial v}{\partial y} - \frac{\partial u}{\partial y}\frac{\partial v}{\partial x}} \begin{bmatrix}\frac{\partial v}{\partial y} & -\frac{\partial u}{\partial y} \\ -\frac{\partial v}{\partial x} & \frac{\partial u}{\partial x}\end{bmatrix}$$

现在就已经直接求出了$\dfrac{\partial z}{\partial u}$与$\dfrac{\partial z}{\partial v}$（等式右侧的所有内容都是已知的）。如果在光源空间中的$\boldsymbol{uv}$平面内移动了$(\Delta u, \Delta v)$个单位，则光源空间的深度值应相应地移动$\Delta z = \Delta u \dfrac{\partial z}{\partial u} + \Delta v \dfrac{\partial z}{\partial v}$。

因此，如果采用这种方法，我们就不必将坐标变换到屏幕空间，而是使它们一直保持在光源空间之中即可——原因是当u或v发生改变时，我们就能直接指出深度值的变化情况。而根据前一节中所述的方案，我们却只能随屏幕空间中x与y坐标的改变而求出具体的深度值。

20.6　小结

1．渲染目标并非后台缓冲区的专利，我们还可以将数据渲染至另一种与之不同的纹理之中。渲染到纹理技术为GPU动态更新纹理的内容提供了一种高效的方式。在将数据渲染到一个纹理之后，我们就可以将此纹理绑定为着色器的输入，并将其映射到几何体之上。渲染到纹理技术可以广泛地应用到多种特效之中，像阴影图、水体仿真以及GPU通用编程技术。

2．如果采用正交投影技术，则视景体便是一个宽度为w、高度为h、近平面为n而远平面为f的长方体，并且投影线皆平行于观察空间中的z轴。这种投影主要用于3D科学程序或工程应用之中，这几种领域大多期望在投影之后平行线继续保持平行（而不像是透视投影那样汇于灭点）。本章的例移就是通过正交投影来模拟平行光所生成的阴影。

3．投影纹理贴图的称谓来源于这种技术可以使我们将纹理投影到任意的几何体之上，这一过程就如同投影机的工作原理。投影纹理贴图的关键在于为每个像素生成相应的纹理坐标，使该纹理看上去似乎是投射到了目标几何体之上。这种纹理坐标被称为投影纹理坐标（projective texture coordinate）。我们先通过将像素投射到投影机的投影平面，而后再将它映射到纹理坐标系内，以此获取其相应的投影纹理坐标。

4．阴影贴图是一种将阴影投射在任意几何体（即不仅限于平面阴影）上的实时阴影渲染技术。阴影贴图的思路是以光源视角将场景的深

度信息渲染至阴影图中。这样一来，阴影图将存有从光源角度来看的可见像素的深度值。随后，我们从摄像机的视角再渲染一遍场景，并使用投影纹理贴图技术将阴影图投射到场景之中。设$s(p)$为从阴影图投射到像素p的深度值，$d(p)$为从光源到该像素的深度，那么，如果$d(p) > s(p)$，则像素p位于阴影之中。即如果像素的深度值$d(p)$大于该像素的阴影图投射深度$s(p)$，则在光源至像素p这条路径中，必存在一像素更接近于光源并遮挡住像素p，因此p将被投射在阴影之中。

5．使用阴影图最让人头疼的问题就是走样。阴影图存储的是以光源视角来看距离其最近的可见像素的深度值。只可惜阴影图的分辨率有限，因此每个阴影图纹素都对应于场景中的一块区域。因此，阴影图仅是一种从光源角度看去场景深度的离散采样。这会导致像阴影粉刺（shadow acne）这种知名的图像混叠问题。采用图形硬件内部支持的斜率缩放偏移（slope-scaled-bias，可在光栅化渲染状态属性D3D12_RASTERIZER_DESC之中设置此功能）技术修正阴影粉刺是一种常用的策略。阴影图有限的分辨率也会导致阴影边缘的锯齿问题。对此，PCF是一种比较流行的解决方案。除此之外，用于解决走样问题的高级方案还有级联阴影图（cascaded shadow map）与方差阴影图（variance shadow map）等。

20.7　练习

1．编写一个程序，尝试通过透视投影与正交投影两种方式分别将纹理投影到场景之中，从而来模拟投影机的工作效果。

2．修改前一个练习的答案，在程序中使用纹理寻址模式，从此令投影机视景体之外的点不受光照。

3．修改练习1的答案，采用聚光灯作为投影机的光源，借此令聚光灯圆锥体照射范围之外的点不会被投影机发出的光照射到。

4．以透视投影代替本章演示程序中的投影方式。需要注意的是，虽然斜率缩放偏移技术可应用于正交投影，但对透视投影来说效果却不是很好。在使用透视投影时能够发现深度图因过度偏移而转为白色（1.0）。试根据图5.25所示的曲线图对此现象做出解释。

5．尝试4096×4096、1024×1024、512×512和256×256这几种不同分辨率的阴影图。

6．试推导出可以实现$[l,r]\times[b,t]\times[n,f]\rightarrow[-1,1]\times[-1,1]\times[0,1]$的长方体映射矩阵。得到的将是一种“偏离中心”（off center）的正交视景体（即此长方体的中心点不在观察空间的原点处）。相对地，20.2节中所推导出的正交投影矩阵变换得到的是一种“位于中心”（on center）的正交视景体。

7．在第17章中，我们曾学习过一种与拾取技术相关的透视投影矩阵。现在试为“偏离中心”的正交投影推导相应的拾取公式。

8．试以单次点采样阴影检测来修改“Shadows”（阴影）演示程序（即不采用PCF）。我们将欣赏到硬阴影（hard shadow）与锯齿状的阴影边缘。

9．关闭斜率缩放偏移来观察阴影粉刺。

10．将斜率缩放偏移值修改为极大的偏移量，以此来观察peter panning失真的效果。

11．正交投影可用于为方向光光源生成阴影图，而透视投影可为聚光灯光源生成阴影图。试解释怎样借助立方体图（cube map）与6个视场角为90°（水平视场角与垂直视场角皆为90°）的透视投影来生成点光源的阴影图。

提示

（请回顾第18章动态中立方体图的生成过程）思考怎样通过立方体图来进行阴影图检测。

[1]　也有译作混叠、锯齿等。我这里为了根据视觉效果分类，更倾向把图像中的波纹作混叠（混叠词大多运用在信号处理方面，这里为区分走样效果而作）、边缘称锯齿，总称走样。

第21章　环境光遮蔽

由于性能的限制，实时光照模型往往会忽略间接光因素（即场景中其他物体所反弹的光线）。但在现实生活中，大部分光照其实是间接光。在第8章中，我们为光照方程引入了环境光项：

$$\boldsymbol{c}_{\mathrm{a}} = \boldsymbol{A}_{\mathrm{L}} \otimes \boldsymbol{m}_{\mathrm{d}}$$

颜色$\boldsymbol{A}_L$表示的是从某光源发出，经环境反射而照射到物理表面的间接光（即环境光）总量。漫反射反照率$\boldsymbol{m}_d$则指出了物体表面根据漫反射率将入射光反射回的总量。所有的环境光项都会以同样的亮度将物体稍微照亮一些，以至阴影中的物体并不是纯黑色的——毕竟我们进行的并非真正的物理计算。间接光会在场景中散射且反射多次，并从各个方向均等地照射在物体之上。图21.1所示的就是这种情况，如果仅采用环境光项来绘制模型，那么物体将会被同一种单一颜色所渲染。

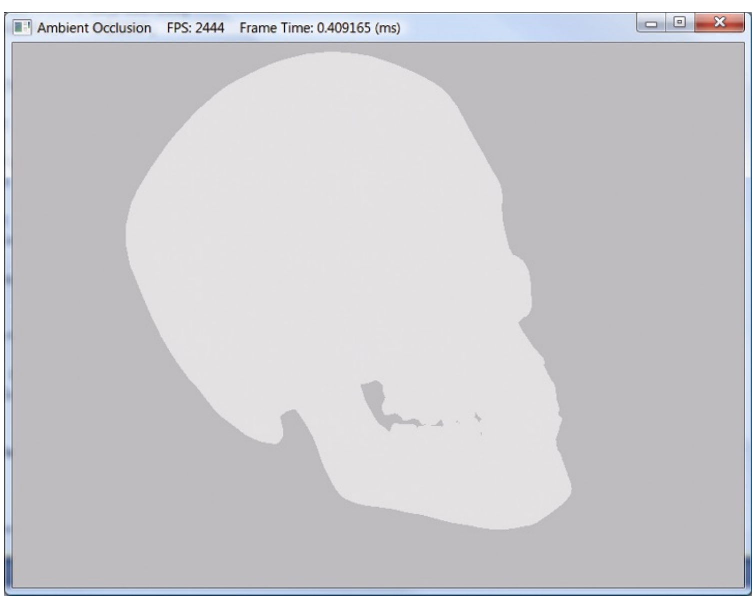


图21.1　这是一个仅用环境光项渲染的网格，整体上只表现出了唯一一种单色

从图21.1可以很明显地看出，我们对环境光项还有一些改良的余地。在本章中，我们将就流行于改善环境光项的环境光遮蔽技术展开讨论。

学习目标：

1．理解环境光遮蔽技术背后的基本原理，并知道如何通过投射光线来实现环境光遮蔽。

2．学习如何在屏幕空间中实现名为“屏幕空间环境光遮蔽”这种近似于实时环境光遮蔽的技术。

21.1　通过投射光线实现环境光遮蔽

环境光遮蔽（ambient occlusion，也有译作环境光吸收等）技术的主体思路如图21.2所示，表面上一点$\boldsymbol{p}$所收到的间接光总量，与照射到以$\boldsymbol{p}$为中心的半球的入射光量成正比。

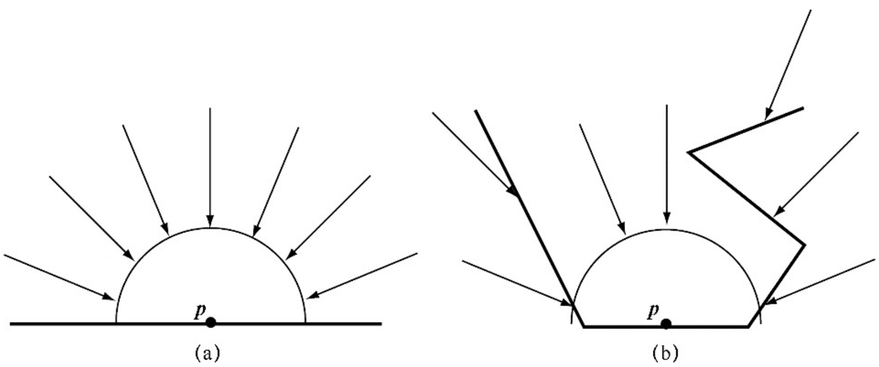


图21.2　（a）中，一个完全不受任何物体遮挡的点$\boldsymbol{p}$，以及所有入射光均能照射到的以$\boldsymbol{p}$为中心的半球。我们以此半球来度量点$\boldsymbol{p}$收到的光量　（b）中，周围环境中的部分几何体遮挡了点$\boldsymbol{p}$，并阻止光线照射到以$\boldsymbol{p}$为中心的半球

一种估算点$\boldsymbol{p}$受遮蔽程度的方法是采用投射光线法（ray casting，从几何角度来看也有译作射线投射法）。具体做法是，我们随机投射出一些光线，使它们穿过以点$\boldsymbol{p}$为中心的半球，并检测这些光线与网格（也就是周围阻挡光线照射到点p的“障碍物”）相交的情况（见图21.3）。如果投射了N条光线，而其中的h条与网格相交，则点$\boldsymbol{p}$所对应的遮蔽率为：

$$\text{occlusion} = \frac{h}{N} \in [0, 1]$$

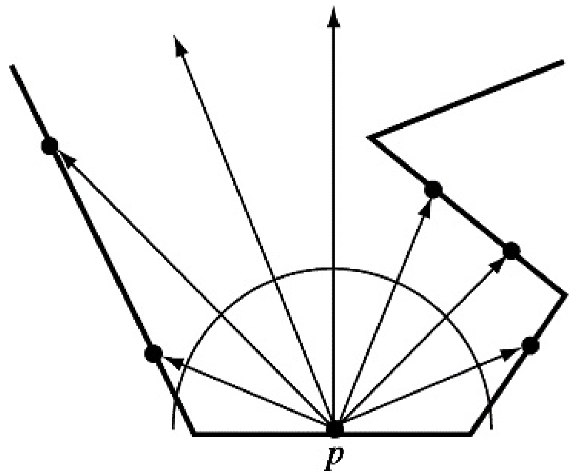


图21.3　通过投射光线来估算环境光遮蔽

事实上，只有当光线与网格的交点$\boldsymbol{q}$到点$\boldsymbol{p}$之间的距离小于某个阈值d时，才会将此光线记作受到遮挡。这是因为若交点$\boldsymbol{q}$与点$\boldsymbol{p}$之间的距离过远就说明在这个方向上照射到点$\boldsymbol{p}$的光不会受到周围物体的遮挡。

遮蔽因子（occlusion factor）用于计量目标点被遮蔽的光线比例（即有多少光线无法接收到）。计算此值的目的，其实是为了用与其意义刚好相反的数据来进行后续的计算工作。即，我们希望了解到底有多少光线可以抵达目标点——这被称为可及率（accessibility，也有译作可访问性等，或称为环境光可及率，ambient-access），此值可根据遮蔽率求出：

$$\text{accessiblity} = 1 - \text{occlusion} \in [0, 1]$$

在以下代码中，我们针对每个网格三角形都投射了光线，并为三角形共用顶点处的遮蔽率求取了平均值。光线的起点就位于三角形的重心（centroid），我们以此生成随机方向的光线通过包围该三角形的半球。

```
void AmbientOcclusionApp::BuildVertexAmbientOcclusion(
  std::vector<Vertex::AmbientOcclusion>& vertices,
  const std::vector<UINT>& indices)
{
  UINT vcount = vertices.size();
  UINT tcount = indices.size()/3;

  std::vector<XMFLOAT3> positions(vcount);
  for(UINT i = 0; i < vcount; ++i)
    positions[i] = vertices[i].Pos;

  Octree octree;
  octree.Build(positions, indices);

  // 对于每个顶点，要统计它们被三角形所共用的情况
  std::vector<int> vertexSharedCount(vcount);

  // 针对每个网格三角形投射光线，并根据共享顶点的三角形数量，对该顶点处的遮蔽数据
求取平均值
  for(UINT i = 0; i < tcount; ++i)
  {
    UINT i0 = indices[i*3+0];
    UINT i1 = indices[i*3+1];
    UINT i2 = indices[i*3+2];

    XMVECTOR v0 = XMLoadFloat3(&vertices[i0].Pos);
    XMVECTOR v1 = XMLoadFloat3(&vertices[i1].Pos);
    XMVECTOR v2 = XMLoadFloat3(&vertices[i2].Pos);

    XMVECTOR edge0 = v1 - v0;
    XMVECTOR edge1 = v2 - v0;

    XMVECTOR normal = XMVector3Normalize(
      XMVector3Cross(edge0, edge1));

    XMVECTOR centroid = (v0 + v1 + v2)/3.0f;

    // 为避免自相交（self intersection）的情况，对重心稍作偏移
    centroid += 0.001f*normal;

    const int NumSampleRays = 32;
    float numUnoccluded = 0;
    for(int j = 0; j < NumSampleRays; ++j)
    {
      XMVECTOR randomDir = MathHelper::RandHemisphereUnitVec3(normal);

      // 测试随机投射出的光线是否与场景中的网格相交
```

```
      // 待办事项：从技术上讲，我们不应当统计距被遮挡三角形过远的交点，但是这样
做对于演示程序来
      // 讲影响不大
      if( !octree.RayOctreeIntersect(centroid, randomDir) )
      {
        numUnoccluded++;
      }
    }

    float ambientAccess = numUnoccluded / NumSampleRays;

    // 为此三角形上的顶点累加环境光可及率，并增加顶点被三角形所引用的次数
    vertices[i0].AmbientAccess += ambientAccess;
    vertices[i1].AmbientAccess += ambientAccess;
    vertices[i2].AmbientAccess += ambientAccess;

    vertexSharedCount[i0]++;
    vertexSharedCount[i1]++;
    vertexSharedCount[i2]++;
  }

  // 最后，通过每个顶点累加的环境光可及率除以累加的顶点共享次数来求取每个顶点处遮
蔽数据的平均值，
  // 并将结果存入顶点属性
  for(UINT i = 0; i < vcount; ++i)
  {
    vertices[i].AmbientAccess /= vertexSharedCount[i];
  }
}
```

注意 Note

例程使用了一棵八叉树（octree）来为光线（射线）与三角形的相交检测提速。对于一个具有上千个三角形的网格来讲，要为每个随机光线与每个网格三角形进行检测，其过程是极其缓慢的。八叉树会将三角形按空间进行划分，因此我们就可以快速地找到那些与光线相交概率更大的三角形，从而大幅减少光线与三角形相交检测的次数。八叉树是一

种经典的空间数据结构，在本章练习1的指引下，我们将继续对其进行探索。

图21.4所示的是仅以上述算法（场景中是没有光源的，即仅用间接光）生成的环境光遮蔽数据所渲染出的模型效果。生成环境光遮蔽的数据作为程序初始化期间的一个预计算步骤，得到的结果存为顶点属性。正如我们所看到的，这个效果与图21.1所示的画面相比有了极大的改观——此时，这个模型才富有立体感。

环境光遮蔽的预计算工作对于静态模型来讲是很易实现的，甚至有一些工具还能直接生成环境光遮蔽图（ambient occlusion map），即存有环境光遮蔽数据的纹理。然而，对于动态模型来讲，这些静态方法就完全不适用了。如果加载并运行了“Ambient Occlusion”（环境光遮蔽）演示程序[1]，我们将发现，仅为一个模型预计算环境光遮蔽数据就要花费若干秒。因此，以光线投射法在运行时实现动态环境光遮蔽技术是不可行的。在下一节中，我们将考察一种通过屏幕空间信息来实时计算环境光遮蔽的流行技术。

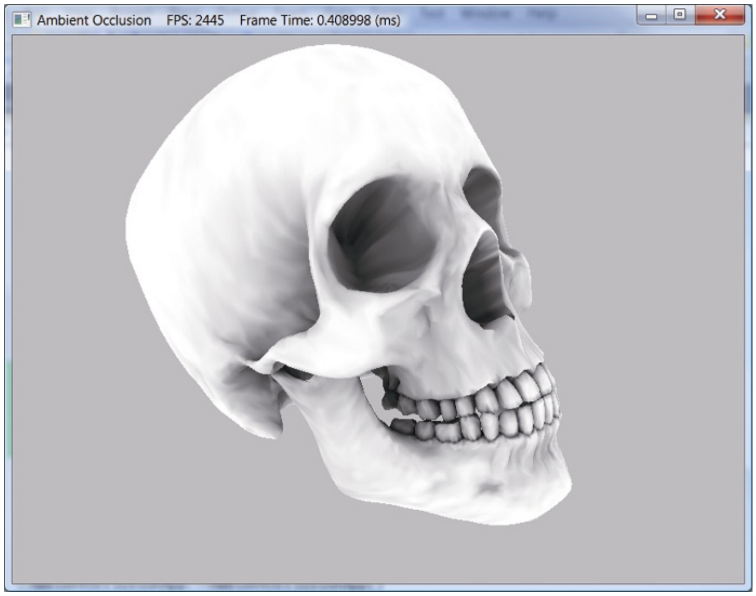


图21.4　仅采用环境光遮蔽技术渲染的网格——其中并不存在任何场景光源。可以注意到模型缝隙间的颜色更深，这是由于从这些地方投射出的光线有更大概率与附近的几何体相交，并增大遮蔽率。此外，天灵盖部分则呈现白色（未受遮蔽），这是因为在我们从此区域上的点向半球投射光线时，它们不会与骷髅头上任何其他的几何体相交

21.2 屏幕空间环境光遮蔽

屏幕空间环境光遮蔽（Screen Space Ambient Occlusion，SSAO）技术所采用的策略是，在渲染每一帧画面的过程中，将场景观察空间中的法线绘制到一个全屏渲染目标（full screen render target），并把场景深度绘制到一个普通的深度/模板缓冲区。接下来，仅用上述观察空间法线渲染目标和深度/模板缓冲区作为输入，在每个像素处估算出相应的环境光遮蔽数据。只要得到了存有每个像素处环境光遮蔽数据的纹理，我们就以这一纹理中的SSAO信息来为每个像素调整环境光项，再像往常那样将处理后的场景绘制到后台缓冲区中。

21.2.1 法线与深度值的渲染过程

首先，我们要把场景中各物体的观察空间法向量渲染到与屏幕大小相同、格式为DXGI_FORMAT_R16G16B16A16_FLOAT的纹理图之内，同时还要绑定置有场景深度的普通深度/模板缓冲区。在这个渲染过程中，所用的顶点着色器与像素着色器的代码如下。

```
// 包含公用的HLSL代码
#include "Common.hlsl"

struct VertexIn
{
  float3 PosL   : POSITION;
  float3 NormalL : NORMAL;
  float2 TexC   : TEXCOORD;
  float3 TangentU : TANGENT;
};
```

```
struct VertexOut
{
  float4 PosH    : SV_POSITION;
  float3 NormalW : NORMAL;
  float3 TangentW : TANGENT;
  float2 TexC    : TEXCOORD;
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout = (VertexOut)0.0f;

  // 获取材质数据
  MaterialData matData = gMaterialData[gMaterialIndex];

  // 这里执行的是等比缩放，否则应使用世界矩阵的逆转置矩阵进行计算
  vout.NormalW = mul(vin.NormalL, (float3x3)gWorld);
  vout.TangentW = mul(vin.TangentU, (float3x3)gWorld);

  // 将顶点变换到齐次裁剪空间
  float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
  vout.PosH = mul(posW, gViewProj);

  // 为三角形插值而输出顶点属性
  float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
  vout.TexC = mul(texC, matData.MatTransform).xy;

  return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
  // 获取材质数据
  MaterialData matData = gMaterialData[gMaterialIndex];
  float4 diffuseAlbedo = matData.DiffuseAlbedo;
  uint diffuseMapIndex = matData.DiffuseMapIndex;
  uint normalMapIndex = matData.NormalMapIndex;

  // 动态查找数组中的纹理
  diffuseAlbedo *= gTextureMaps[diffuseMapIndex].
    Sample(gsamAnisotropicWrap, pin.TexC);

#ifdef ALPHA_TEST
  // 丢弃纹理中alpha < 0.1的像素。我们应在着色器中及早地进行这项测试，尽量满足条
件的像素提前
  // 退出着色器的处理，跳过后续没必要执行的着色器代码，从而优化性能
  clip(diffuseAlbedo.a - 0.1f);
#endif
```

```
  // 对法线插值可能导致其非规范化，因此要重新对它进行规范化处理
  pin.NormalW = normalize(pin.NormalW);

  // 注意：为SSAO而使用插值顶点法线

  // 返回法线的观察空间坐标
  float3 normalV = mul(pin.NormalW, (float3x3)gView);
  return float4(normalV, 0.0f);
}
```

如上述代码所示，像素着色器输出了观察空间中的法向量。另外，这一次采用的是浮点渲染目标，因此向其中写入任何浮点数据都是合理的。

21.2.2　环境光遮蔽的渲染过程

在布置好观察空间法线以及场景深度之后，我们就禁用深度缓冲区（在生成环境光遮蔽纹理时不需要对该缓冲区做任何改动），并在每个像素处调用SSAO像素着色器来绘制一个全屏的四边形（full screen quad）。这样一来，该像素着色器将运用法线纹理以及深度缓冲区为每个像素生成一个对应的环境光可及率数据，我们称在这个渲染过程中所生成的纹理图为**SSAO图**（SSAO map）。尽管我们按照全屏分辨率（即后台缓冲区的分辨率）渲染了法线图与深度图，但是出于对性能的考虑，仅按深度缓冲区高度以及宽度的一半来渲染SSAO图。以这种方式来渲染SSAO图并不会过于影响渲染质量，因为环境光遮蔽本身就是一种低频效果（low frequency effect，LFE）[2]。注意，后续几小节都是围绕图21.5展开的。

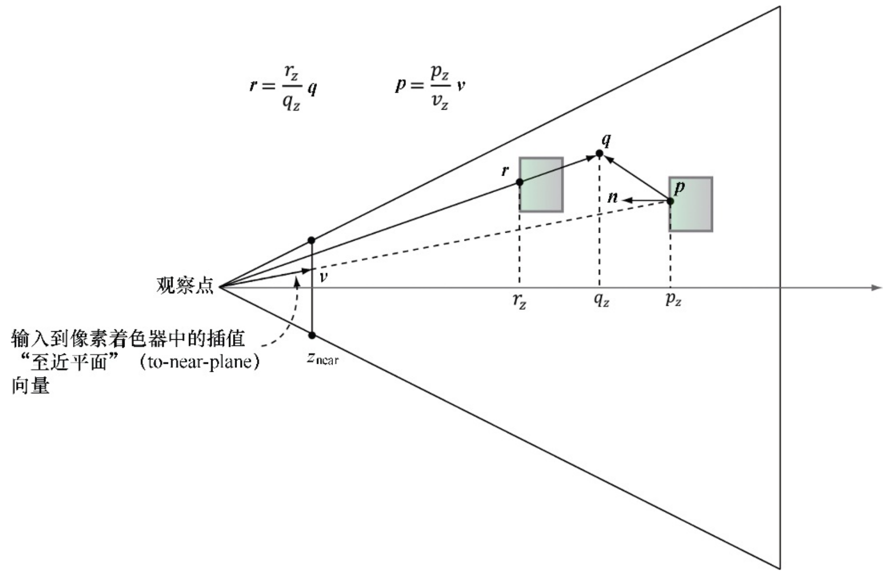


图21.5　SSAO技术中要用到的各种关键点。点$\boldsymbol{p}$就是我们当前正在处理的像素，根据从观察点至该像素在近平面内对应点的向量$\boldsymbol{v}$以及深度缓冲区中存储的对应深度值来重新构建点$\boldsymbol{p}$。点$\boldsymbol{q}$是以点$\boldsymbol{p}$为中心的半球内的随机一点，点$\boldsymbol{r}$则是从观察点到$\boldsymbol{q}$这一路径上的最近可视点。如果$|p_z - r_z|$足够小，且$\boldsymbol{r} - \boldsymbol{p}$与$\boldsymbol{n}$之间的夹角小于$90°$，那么点$\boldsymbol{r}$将计入点$\boldsymbol{p}$的遮蔽值。在这个例程中，我们采用了14个随机样点，再根据平均值法求得的遮蔽率来估算屏幕空间中的环境光遮蔽数据

21.2.2.1　重新构建待处理点在观察空间中的位置

当我们为绘制全屏四边形而对SSAO图中的每个像素依次调用SSAO像素着色器时，可利用投影矩阵的逆矩阵，将位于NDC空间中四边形的角点（corner point）变换到近平面投影窗口上的点。

```
static const float2 gTexCoords[6] =
{
  float2(0.0f, 1.0f),
  float2(0.0f, 0.0f),
  float2(1.0f, 0.0f),
  float2(0.0f, 1.0f),
  float2(1.0f, 0.0f),
  float2(1.0f, 1.0f)
};
```

```
// 利用构成四边形的6个顶点进行绘制调用
VertexOut VS(uint vid : SV_VertexID)
{
  VertexOut vout;

  vout.TexC = gTexCoords[vid];

  // 将展示在屏幕上全屏四边形变换至NDC空间
  vout.PosH = float4(2.0f*vout.TexC.x - 1.0f, 1.0f - 2.0f*vout.TexC.y,
    0.0f, 1.0f);

  // 将四边形的各角点变换到观察空间的近平面
  float4 ph = mul(vout.PosH, gInvProj);
  vout.PosV = ph.xyz / ph.w;

  return vout;
}
```

这些“至近平面”（to-near-plane）向量都是经四边形内插而得到的，它给出的图21.5中给出的$\boldsymbol{v}$就是每个像素从观察点到近平面的向量。现在，我们将为每个像素采集深度值以获取点$\boldsymbol{p}$至观察点路径中最近可视点z坐标p_z的NDC坐标。这样做的最终目的，是根据采集的NDC空间中的z坐标p_z以及插值“至近平面”向量$\boldsymbol{v}$，来重新构建$\boldsymbol{p}$在观察空间中的位置$\boldsymbol{p}=(p_x, p_y, p_z)$。以下是重建的思路：由于与向量$\boldsymbol{v}$同起点共方向的光线通过点$\boldsymbol{p}$，因此也就存在一个值$t$满足$\boldsymbol{p}=t\boldsymbol{v}$。此时，我们得到$p_z=tv_z$，那么$t=p_z/v_z$。因此$\boldsymbol{p}=\frac{p_z}{v_z}\boldsymbol{v}$。像素着色器中重建观察空间位置的代码如下。

```
float NdcDepthToViewDepth(float z_ndc)
{
  // 我们可以执行将z坐标从NDC空间变换到观察空间的逆运算。由于我们有z_ndc = A + 
B/viewZ，
  // 其中gProj[2,2]=A且gProj[3,2]=B，因此……
  float viewZ = gProj[3][2] / (z_ndc - gProj[2][2]);
  return viewZ;
}
```

```
float4 PS(VertexOut pin) : SV_Target
{
  // 从深度图中获取该像素在NDC空间内的z坐标
  float pz = gDepthMap.SampleLevel(gsamDepthMap, pin.TexC, 0.0f).r;
  // 将深度值变换到观察空间
  pz = NdcDepthToViewDepth(pz);

  // 用深度值pz 重新构建此点在观察空间中的位置
  float3 p = (pz/pin.PosV.z)*pin.PosV;

  [...]
}
```

21.2.2.2 生成随机样点

这个步骤模拟的是向半球随机投射光线的过程。我们以$\boldsymbol{p}$为中心，在指定的遮蔽半径（occlusion radiu）内随机地从点$\boldsymbol{p}$的前侧部分采集N个点，并将其中的任意一点记作$\boldsymbol{q}$。遮蔽半径是一项影响艺术效果的参数，它控制着我们采集的随机样点相对于点$\boldsymbol{p}$的距离。而选择仅采集点$\boldsymbol{p}$前侧部分的点，就相当于在以光线投射的方式执行环境光遮蔽时，只需在半球内进行投射而不必在完整的球体内投射而已。

接下来的问题是如何来生成随机样点。一种解决方案是，我们可以生成随机向量并将它们存于一个纹理图中，再在纹理图的N个不同位置获取N个随机向量。然而，由于整个计算过程都是随机的，所以我们并不能保证采集的向量必定是均匀分布，也就是说，会有全部向量趋于同向的风险，如此一来，遮蔽率的估算结果必然有失偏颇。为了解决这个问题，我们将采取下列技巧。在我们采用的实现方法之中共使用了$N = 14$个采样点，并以下列C++代码生成14个均匀分布的向量。

```
void Ssao::BuildOffsetVectors()
```

```
{
  // 采用14个均匀分布的向量实现环境光遮蔽技术。我们选择的是立方体的8个角点以及6个面上的中心
  // 点作为向量点。并将这些点以立方体空间位置上相对的顺序交替排列。这样一来，即使选用的采样点小
  // 于14个，我们仍然可以得到比较分散的向量

  // 8个立方体角点
  mOffsets[0] = XMFLOAT4(+1.0f, +1.0f, +1.0f, 0.0f);
  mOffsets[1] = XMFLOAT4(-1.0f, -1.0f, -1.0f, 0.0f);

  mOffsets[2] = XMFLOAT4(-1.0f, +1.0f, +1.0f, 0.0f);
  mOffsets[3] = XMFLOAT4(+1.0f, -1.0f, -1.0f, 0.0f);

  mOffsets[4] = XMFLOAT4(+1.0f, +1.0f, -1.0f, 0.0f);
  mOffsets[5] = XMFLOAT4(-1.0f, -1.0f, +1.0f, 0.0f);

  mOffsets[6] = XMFLOAT4(-1.0f, +1.0f, -1.0f, 0.0f);
  mOffsets[7] = XMFLOAT4(+1.0f, -1.0f, +1.0f, 0.0f);

  // 6个立方体面的中心点
  mOffsets[8] = XMFLOAT4(-1.0f, 0.0f, 0.0f, 0.0f);
  mOffsets[9] = XMFLOAT4(+1.0f, 0.0f, 0.0f, 0.0f);

  mOffsets[10] = XMFLOAT4(0.0f, -1.0f, 0.0f, 0.0f);
  mOffsets[11] = XMFLOAT4(0.0f, +1.0f, 0.0f, 0.0f);

  mOffsets[12] = XMFLOAT4(0.0f, 0.0f, -1.0f, 0.0f);
  mOffsets[13] = XMFLOAT4(0.0f, 0.0f, +1.0f, 0.0f);

  for(int i = 0; i < 14; ++i)
  {
    // 创建长度范围在[0.25, 1.0]内的随机长度向量
    float s = MathHelper::RandF(0.25f, 1.0f);

    XMVECTOR v = s * XMVector4Normalize(XMLoadFloat4(&mOffsets[i]));

    XMStoreFloat4(&mOffsets[i], v);
  }
}
```

注意 Note

因为使用的是4D齐次向量，所以在HLSL文件中设置偏移向量数组（offset vector array，即上文C++部分中的`mOffsets`与下文HLSL文件中的`gOffsetVectors`）时，不必担心任何的对齐问题。

到目前为止，我们在像素着色器中采集了一次随机向量的纹理图，并用它来对14个均匀分布的向量进行反射（reflect）。其最终结果便是获得了14个均匀分布的随机向量（equally distributed random vector）。

21.2.2.3 生成潜在的遮蔽点

我们现在获得了围绕点$\mathbf{p}$的各随机采样点$\mathbf{q}$。由于仍不知道它们所处的位置到底是空无一物还是实心物体，因此还不足以检测出在这个方向上点$\mathbf{p}$是否被遮住了。为了找到那些有可能遮挡住点$\mathbf{p}$的点，就需要用到深度缓冲区中的深度信息。所以我们要做的就是以摄像机的视角为每个点$\mathbf{q}$生成投影纹理坐标，根据这些坐标来对深度缓冲区采样以获取NDC空间中对应的深度值，接着将它们变换至观察空间来求得从观察点至点$\mathbf{p}$方向上距离观察点最近的深度值r_z。知道了z坐标r_z后，我们就能依照类似于21.2.2.1节中所述的方法，重新构建出全屏3D观察空间中点$\mathbf{r}$的准确位置。由于从观察点至点$\mathbf{q}$方向的向量会经过点$\mathbf{r}$，因此存在t使得$\mathbf{r}=t\mathbf{q}$。特别是$r_z=tq_z$，那么$t=r_z/q_z$。所以，$\mathbf{r}=\frac{r_z}{q_z}\mathbf{q}$。由此可知，根据每个随机采样点$\mathbf{q}$所生成的点$\mathbf{r}$即为潜在的遮蔽点。

21.2.2.4 执行遮蔽检测

既然我们现在得到了潜在的遮蔽点$\boldsymbol{r}$，也就能够执行遮蔽检测来估算点$\boldsymbol{p}$是否被它们遮挡住。该测试依赖于两种量值。

1．观察空间中点$\boldsymbol{p}$与点$\boldsymbol{r}$的深度距离为$|p_z - r_z|$。随着此距离的增长，遮蔽值将按比例线性缩小。这是因为随着遮蔽点与目标点距离越远，其遮蔽效果也就越弱。如果该距离超过某个指定的最大距离，那么点$\boldsymbol{r}$将完全不会遮挡点$\boldsymbol{p}$。而且，如果此距离过小，我们就认为点$\boldsymbol{p}$与点$\boldsymbol{q}$位于同一平面上（共面），因此点$\boldsymbol{q}$在这种情况下也不会遮挡点$\mathbf{p}$。

2．向量$\boldsymbol{n}$与$\boldsymbol{r}-\boldsymbol{p}$之间夹角的测定方法为$\max\left(\boldsymbol{n}\cdot\left(\frac{\boldsymbol{r}-\boldsymbol{p}}{\|\boldsymbol{r}-\boldsymbol{p}\|}\right),0\right)$。这是为防止自相交（self-intersection）情况的发生（见图21.6）。

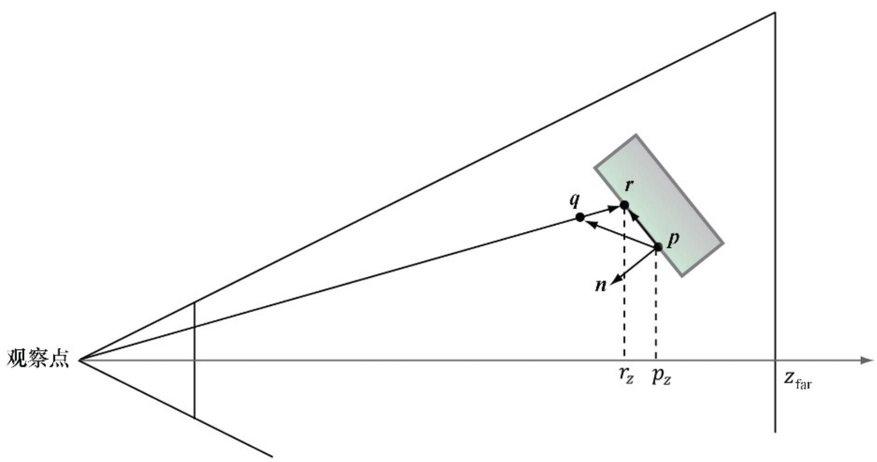


图21.6　如果点$\boldsymbol{r}$与点$\boldsymbol{p}$位于同一平面内，便可满足第一个条件，即距离$|p_z - r_z|$足够小以至于点$\boldsymbol{r}$遮蔽了点$\boldsymbol{p}$。然而，从图中可以看出，两者在同一平面内的时候，点$\boldsymbol{r}$并没有遮挡点$\boldsymbol{p}$。通过计算$\max\left(\boldsymbol{n}\cdot\left(\frac{\boldsymbol{r}-\boldsymbol{p}}{\|\boldsymbol{r}-\boldsymbol{p}\|}\right),0\right)$调整遮蔽值便可以防止对此情况的误判

21.2.2.5　完成计算过程

在将每个样点的遮蔽数据相加之后，还要通过除以采样的次数来计算遮蔽率。接着，我们会计算环境光可及率，并对它进行幂运算以提高对比度（contrast）。当然，我们也能够按需求适当增加一些数值来提高光照强度，以此为环境光图（ambient map）增加亮度。除此之外，我们还可以尝试不同的对比值与亮度值。

```
occlusionSum /= gSampleCount;

float access = 1.0f - occlusionSum;

// 提高SSAO图的对比度，使SSAO的效果更为显著
return saturate(pow(access, 4.0f));
```

21.2.2.6　具体实现

在前一节中，我们罗列了生成SSAO图的关键要点。以下是HLSL程序的具体实现。

```
//=====================================================================
// Ssao.hlsl的作者为Frank Luna (C) 2015版权所有
//=====================================================================

cbuffer cbSsao : register(b0)
{
  float4x4 gProj;
  float4x4 gInvProj;
  float4x4 gProjTex;
  float4  gOffsetVectors[14];

  // 用于SsaoBlur.hlsl程序
  float4 gBlurWeights[3];

  float2 gInvRenderTargetSize;

  // 指定的观察空间中的各坐标
  float  gOcclusionRadius;
  float  gOcclusionFadeStart;
  float  gOcclusionFadeEnd;
```

```
  float  gSurfaceEpsilon;
};

cbuffer cbRootConstants : register(b1)
{
  bool gHorizontalBlur;
};

// 非数值数据是不能添加到常量缓冲区中的
Texture2D gNormalMap  : register(t0);
Texture2D gDepthMap   : register(t1);
Texture2D gRandomVecMap : register(t2);

SamplerState gsamPointClamp : register(s0);
SamplerState gsamLinearClamp : register(s1);
SamplerState gsamDepthMap : register(s2);
SamplerState gsamLinearWrap : register(s3);

static const int gSampleCount = 14;

static const float2 gTexCoords[6] =
{
  float2(0.0f, 1.0f),
  float2(0.0f, 0.0f),
  float2(1.0f, 0.0f),
  float2(0.0f, 1.0f),
  float2(1.0f, 0.0f),
  float2(1.0f, 1.0f)
};

struct VertexOut
{
  float4 PosH : SV_POSITION;
  float3 PosV : POSITION;
  float2 TexC : TEXCOORD0;
};

VertexOut VS(uint vid : SV_VertexID)
{
  VertexOut vout;

  vout.TexC = gTexCoords[vid];

  // 将呈现在屏幕上的全屏四边形变换NDC空间
  vout.PosH = float4(2.0f*vout.TexC.x - 1.0f, 1.0f - 2.0f*vout.TexC.y,
    0.0f, 1.0f);

  // 将四边形的诸角点变换至观察空间中的近平面上
```

```
  float4 ph = mul(vout.PosH, gInvProj);
  vout.PosV = ph.xyz / ph.w;

  return vout;
}

// 将确定目标样点p被点q遮挡程度的任务封装成这个以distZ作为参数的函数
float OcclusionFunction(float distZ)
{
  //
  // 如果depth(q)位于depth(p)“之后”（超出半球范围），则点q无法遮挡点p
  // 而且，若depth(q)与depth(p)距离过近，亦认为点q不能遮住点p，这是因为只有点q
位于点p之
  // 前并根据用户定义的Epsilon值才能确定点q对点p的遮蔽程度
  //
  // 我们通过下列函数来确定遮蔽值
  //
  //
           //   1.0      -------------  \
           //            |            |    \
           //            |            |      \
           //            |            |        \
           //            |            |          \
           //            |            |            \
           //            |            |              \
           // ----|------|-----------|-------------|-------|---> zv
           //     0      Eps          z0              z1
           //

  float occlusion = 0.0f;
  if(distZ > gSurfaceEpsilon)
  {
    float fadeLength = gOcclusionFadeEnd - gOcclusionFadeStart;

    // 随着distZ由gOcclusionFadeStart趋向于gOcclusionFadeEnd，遮蔽值由1线性
减小至0
    occlusion = saturate( (gOcclusionFadeEnd-distZ)/fadeLength );
  }

  return occlusion;
}

float NdcDepthToViewDepth(float z_ndc)
{
  // z_ndc = A + B/viewZ，其中gProj[2,2]=A且gProj[3,2]=B
  float viewZ = gProj[3][2] / (z_ndc - gProj[2][2]);
  return viewZ;
```

```
}

float4 PS(VertexOut pin) : SV_Target
{
  // p -- 我们要计算的环境光遮蔽目标点
  // n -- 点p处的法向量
  // q -- 随机偏离于点p的一点
  // r -- 有可能遮挡点p的一点

  // 获得像素p于观察空间中的法线与z坐标
  float3 n = gNormalMap.SampleLevel(gsamPointClamp, pin.TexC, 0.0f).xyz;
  float pz = gDepthMap.SampleLevel(gsamDepthMap, pin.TexC, 0.0f).r;
  pz = NdcDepthToViewDepth(pz);

  //
  // 重新构建精确的全屏观察空间位置(x,y,z)
  // 求出满足p = t*pin.PosV的t
  // p.z = t*pin.PosV.z
  // t = p.z / pin.PosV.z
  //
  float3 p = (pz/pin.PosV.z)*pin.PosV;

  // 提取随机向量并将它从区间[0,1]映射至[-1, +1]
  float3 randVec = 2.0f*gRandomVecMap.SampleLevel(
    gsamLinearWrap, 4.0f*pin.TexC, 0.0f).rgb - 1.0f;

  float occlusionSum = 0.0f;

  // 在以p为中心的半球内，根据法线n对p周围的点进行采样
  for(int i = 0; i < gSampleCount; ++i)
  {
    // 偏移向量都是固定且均匀分布的（所以我们采用的偏移向量不会在同一方向上扎堆）
。如果将它们关
    // 于一个随机向量进行反射，则得到的必为一组均匀分布的随机偏移向量
    float3 offset = reflect(gOffsetVectors[i].xyz, randVec);

    // 如果此偏移向量位于(p, n)所定义的平面之后，就翻转（flip）该偏移向量
    float flip = sign( dot(offset, n) );

    // 在遮蔽半径内采集靠近点p的点
    float3 q = p + flip * gOcclusionRadius * offset;

    // 投影点q并生成相应的投影纹理坐标
    float4 projQ = mul(float4(q, 1.0f), gProjTex);
    projQ /= projQ.w;

    // 沿着从观察点至点q的光线方向，寻找离观察点最近的深度值。（注意，此值未必是
```

```
点q的深度值，
    // 因为点q只是接近于p的任意一点，而其位置可能是空无一物）。为此，我们就需要查看此点在深度
    // 图中的深度值

    float rz = gDepthMap.SampleLevel(gsamDepthMap, projQ.xy, 0.0f).r;
    rz = NdcDepthToViewDepth(rz);

    // 重新构建观察空间中的位置坐标r = (rx,ry,rz)。我们知道点r位于观察点至点q的光线上，因
    // 此也就存在t满足r = t*q
    // r.z = t*q.z ==> t = r.z / q.z

    float3 r = (rz / q.z) * q;

    //
    // 测试点r是否遮挡着点p
    //  * 点积dot(n, normalize(r - p))度量的是遮蔽点r距平面(p,n)前侧的距离。越趋于此平
    // 面的前侧，我们就给它设定越大的遮蔽权重。同时，这也能够防止位于倾斜面(p,n)上一点r的自阴
    // 影（self shadow）所产生出错误的遮蔽值，这是因为在以观察点的视角来看，它们有着不同的
    // 深度值，但事实上，位于倾斜面(p,n)上的点r却没有遮挡目标点p
    //  * 遮蔽权重的大小依赖于遮蔽点与其目标点之间的距离。如果遮蔽点r离目标点p过远，则认为点
    //r不会遮挡遮蔽点p
    //

    float distZ = p.z - r.z;
    float dp = max(dot(n, normalize(r - p)), 0.0f);
    float occlusion = dp * OcclusionFunction(distZ);

    occlusionSum += occlusion;
  }

  occlusionSum /= gSampleCount;

  float access = 1.0f - occlusionSum;

  // 增强SSAO图的对比度，使SSAO图的效果更加明显
  return saturate(pow(access, 2.0f));
}
```

注意 Note

对于观察距离（viewing distance，也有译作取景距离、视距等）过远的场景来说，由于深度缓冲区精度的限制可能产生具有错误的渲染效果。一种简单的解决方案是随着距离逐渐变远来渐渐模糊SSAO的效果。

21.2.3 模糊过程

图21.7所示的是我们生成的环境光遮蔽图的当前效果。其中的噪点是由于随机样点过少所导致的。但通过采集足够多的样点来屏蔽噪点的做法，在实时渲染的前提下并不切实际。对此，常用的解决方案是采用边缘保留模糊（edge preserving blur，也译作保边模糊。这里采用的为双边模糊，即bilateral blur）的过滤方式来使SSAO图的过渡更为平滑。如果使用的过滤方法为非边缘保留模糊，那么随着物体边缘的明显划分转为平滑渐变，会使场景中的物体难以界定。这种边缘保留模糊的算法与第13章中实现的模糊方法相似，唯一的区别在于需要新添加一个条件语句，以令边缘不受模糊处理（要靠法线图与深度图来检测边缘）。

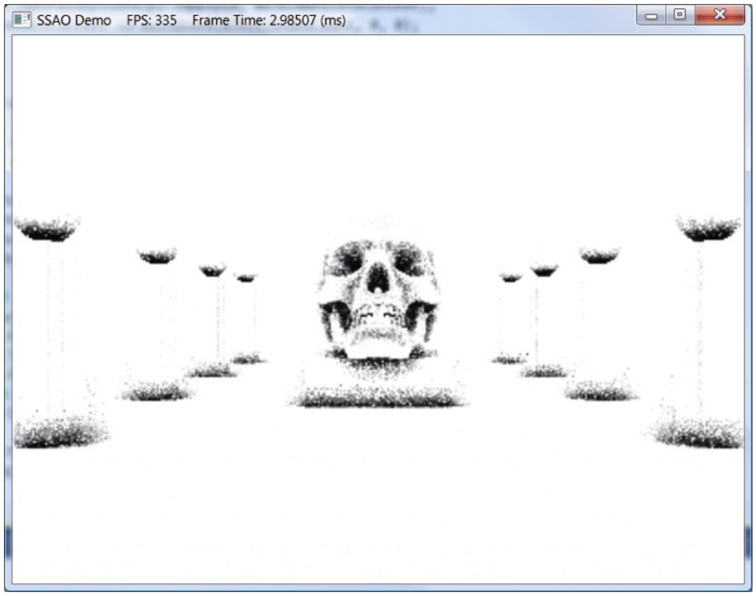


图21.7　由于我们仅采集了少量的随机样本，从而导致SSAO出现了噪点

```
//=====================================================================
// SsaoBlur.hlsl的作者为Frank Luna (C) 2015版权所有
//
// 为环境光图执行双边保边模糊。我们以像素着色器代替计算着色器来避免从计算模式向渲
染模式的转换。
// 纹理缓存（texture cache）适当地弥补了不具共享内存的缺陷。环境光图采用的是16
位的纹理格式，
// 由于它占用空间较小，所以适于在缓存中存储大量纹素
//======================================================================

cbuffer cbSsao : register(b0)
{
  float4x4 gProj;
  float4x4 gInvProj;
  float4x4 gProjTex;
  float4  gOffsetVectors[14];

  // 用于SsaoBlur.hlsl程序
  float4 gBlurWeights[3];

  float2 gInvRenderTargetSize;

  // 给定的观察空间中的诸坐标
  float gOcclusionRadius;
  float gOcclusionFadeStart;
  float gOcclusionFadeEnd;
  float gSurfaceEpsilon;
```

```
};

cbuffer cbRootConstants : register(b1)
{
  bool gHorizontalBlur;
};

// 非数值数据是不能添加到常量缓冲区中的
Texture2D gNormalMap : register(t0);
Texture2D gDepthMap : register(t1);
Texture2D gInputMap : register(t2);

SamplerState gsamPointClamp : register(s0);
SamplerState gsamLinearClamp : register(s1);
SamplerState gsamDepthMap : register(s2);
SamplerState gsamLinearWrap : register(s3);

static const int gBlurRadius = 5;

static const float2 gTexCoords[6] =
{
  float2(0.0f, 1.0f),
  float2(0.0f, 0.0f),
  float2(1.0f, 0.0f),
  float2(0.0f, 1.0f),
  float2(1.0f, 0.0f),
  float2(1.0f, 1.0f)
};

struct VertexOut
{
  float4 PosH : SV_POSITION;
  float2 TexC : TEXCOORD;
};

VertexOut VS(uint vid : SV_VertexID)
{
  VertexOut vout;

  vout.TexC = gTexCoords[vid];

  // 将显示在屏幕上的全屏四边形变换至NDC空间中
  vout.PosH = float4(2.0f*vout.TexC.x - 1.0f, 1.0f - 2.0f*vout.TexC.y,
    0.0f, 1.0f);

  return vout;
}
```

```
float NdcDepthToViewDepth(float z_ndc)
{
  // z_ndc = A + B/viewZ，其中gProj[2,2]=A且gProj[3,2]=B
  float viewZ = gProj[3][2] / (z_ndc - gProj[2][2]);
  return viewZ;
}

float4 PS(VertexOut pin) : SV_Target
{
  // 将模糊权重解包到浮点数组中
  float blurWeights[12] =
  {
    gBlurWeights[0].x, gBlurWeights[0].y, gBlurWeights[0].z,
    gBlurWeights[0].w,
    gBlurWeights[1].x, gBlurWeights[1].y, gBlurWeights[1].z,
    gBlurWeights[1].w,
    gBlurWeights[2].x, gBlurWeights[2].y, gBlurWeights[2].z,
    gBlurWeights[2].w,
  };

    float2 texOffset;
    if(gHorizontalBlur)
    {
        texOffset = float2(gInvRenderTargetSize.x, 0.0f);
    }
    else
    {
        texOffset = float2(0.0f, gInvRenderTargetSize.y);
    }

    // 总是将中心值计入总和之中
    float4 color   = blurWeights[gBlurRadius] * gInputMap.SampleLevel(
      gsamPointClamp, pin.TexC, 0.0);
    float totalWeight = blurWeights[gBlurRadius];

  float3 centerNormal = gNormalMap.SampleLevel(gsamPointClamp, pin.TexC, 0
.0f).xyz;
  float centerDepth = NdcDepthToViewDepth(
    gDepthMap.SampleLevel(gsamDepthMap, pin.TexC, 0.0f).r);

  for(float i = -gBlurRadius; i <= gBlurRadius; ++i)
  {
    // 此前已经计入了中心权重
    if( i == 0 )
      continue;
```

```
    float2 tex = pin.TexC + i*texOffset;

    float3 neighborNormal = gNormalMap.SampleLevel(gsamPointClamp, tex, 0.
0f).xyz;
    float neighborDepth = NdcDepthToViewDepth(
      gDepthMap.SampleLevel(gsamDepthMap, tex, 0.0f).r);

    //
    // 如果中心值与邻近数值相差太大（不论法线还是深度值），就假设正在采集的部分是
不连续的（即处
    // 于物体边缘），继而不对这种样本进行模糊处理
    //

    if( dot(neighborNormal, centerNormal) >= 0.8f &&
      abs(neighborDepth - centerDepth) <= 0.2f )
    {
      float weight = blurWeights[i + gBlurRadius];

      // 累加邻近像素的颜色数据以进行模糊处理
      color += weight*gInputMap.SampleLevel(
        gsamPointClamp, tex, 0.0);

      totalWeight += weight;
    }
  }

  // 使总权重之和为1，以弥补被忽略而未计入统计的样本
  return color / totalWeight;
}
```

图21.8所示的即为经边缘保留模糊处理后的环境光图。

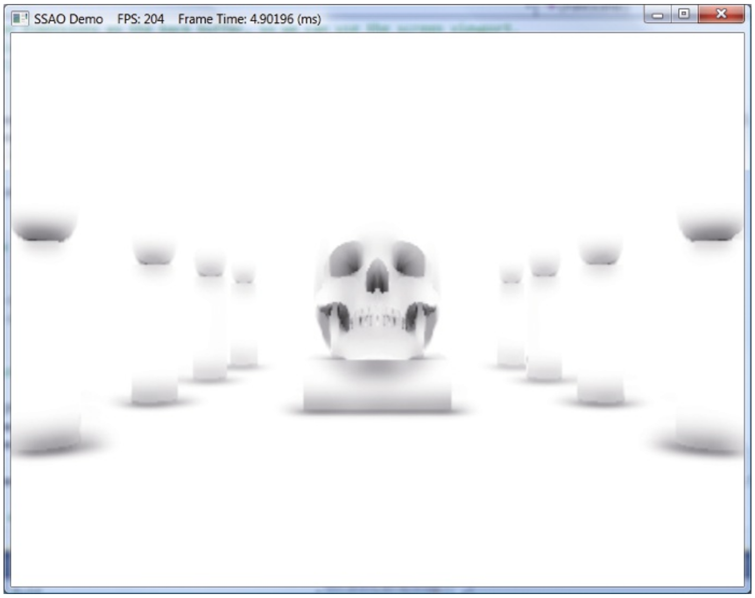


图21.8　经保边模糊对噪点进行平滑处理后的效果。在演示程序中，共对图像进行了3次模糊处理

21.2.4　使用环境光遮蔽图

到此为止，我们已经构造出了环境光遮蔽图，最后的步骤就是将其应用到场景之中。思路之一是使用alpha混合技术，通过后台缓冲区来调整环境光图。可是，一旦照此方案去做，就会发现环境光图要变动的并不只是环境光项，还要波及光照方程中的漫反射项与镜面反射项。由此可见，这种处理方式并不是很妙。因此，现采用如下策略：在将场景渲染到后台缓冲区时，我们要把环境光图作为着色器的输入。接下来再（以摄像机的视角）生成投影纹理坐标，对SSAO图进行采样，并将它应用至光照方程的环境光项。

```
// 在顶点着色器中，为投影场景里的SSAO图而生成投影纹理坐标
vout.SsaoPosH = mul(posW, gViewProjTex);

// 在像素着色器中，完成纹理投影并对SSAO图进行采样
```

```
pin.SsaoPosH /= pin.SsaoPosH.w;
float ambientAccess = gSsaoMap.Sample(gsamLinearClamp, pin.SsaoPosH.xy, 0.0f).r;

// 根据采样数据按比例缩放光照方程中的环境光项
float4 ambient = ambientAccess*gAmbientLight*diffuseAlbedo;
```

图21.9所示的是应用了SSAO图之后的场景效果。SSAO的效果可能并不是十分明显，我们可以在场景中反射充足的环境光，以此提升环境光可及率来令反差更为显著。当物体位于阴影之中时，SSAO的优点则尤为明显：此时，漫反射光项与镜面反射光项纷纷失效，而仅表现出环境光项。若这时不采用SSAO技术，则阴影中的物体会因恒定的环境光项而显得没有立体感，但是采用了SSAO之后，它们仍将保持3D画风。

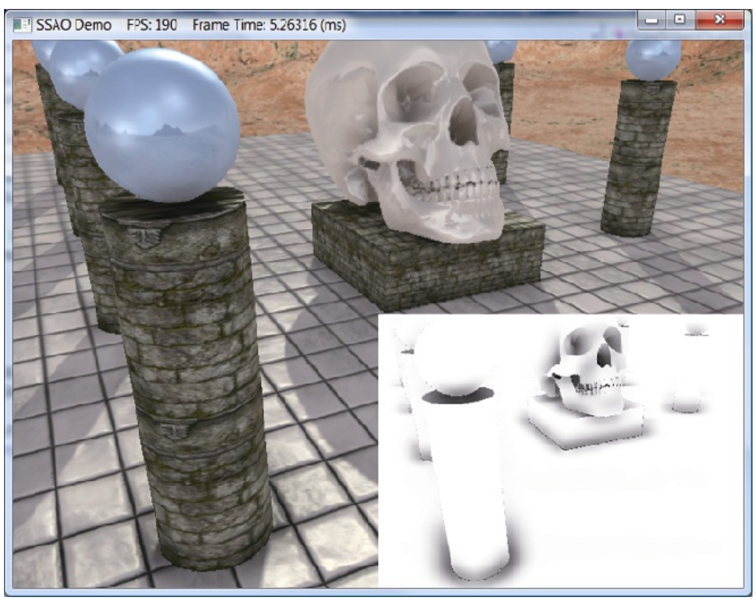


图21.9　本章例程的效果。由于SSAO图仅影响环境光项，所以效果不太明显。但是，我们依然可以从柱体和方盒的底端、球体的下侧以及骷髅头的周围等部分看出稍显偏暗

在渲染观察空间中场景众法线的同时，我们也要为场景构建深度缓冲区。因此，以SSAO图第二次渲染场景时，应将深度检测的比较方法

改为“EQUALS”。由于只有距离观察点最近的可视像素才能通过这项深度比较检测，所以该检测方法就能有效防止第二次渲染过程中的重复绘制（overdraw）操作。而且，在第二次渲染过程中也无须向深度缓冲区执行写操作，这是因为我们已经在法线渲染目标的绘制过程中将场景深度写入了深度缓冲区。

```
opaquePsoDesc.DepthStencilState.DepthFunc = D3D12_COMPARISON_FUNC_EQUAL;
opaquePsoDesc.DepthStencilState.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ZERO;
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(
  &opaquePsoDesc, IID_PPV_ARGS(&mPSOs["opaque"])));
```

21.3 小结

1．光照方程中的环境光项模拟了间接光。在我们所采用光照模型中，环境光项仅仅是一个常量。因此，当物体在阴影之中或者仅有环境光照射其表面时，模型就会因单一颜色体现不出物体实体形状（solid definition）而显得扁平化、不立体。环境光遮蔽技术的目的就是为环境光项找寻一个更佳的估算方法，使得物体只使用环境光项也能富有立体感。

2．间接光遮蔽的实现思路是，物体表面上一点$\boldsymbol{p}$所接收到的间接光量，与照射到以点$\boldsymbol{p}$为中心的半球的入射光量成正比。一种估算点$\boldsymbol{p}$遮蔽率的方式是投射光线：我们从点$\boldsymbol{p}$向以点$\boldsymbol{p}$为中心的半球随机投射光线，并检测它们与周围网格的相交情况。如果这些光线没有与任何几何体相交，那么就认为点$\boldsymbol{p}$完全没有被遮挡住。但是，若存在许多光线与网格相交的现象，我们则认定点$\boldsymbol{p}$被场景中的物体遮挡严重。

3．对于动态物体的实时渲染来讲，通过光线投射法来进行环境光遮蔽过于浪费。相对而言，屏幕空间环境光遮蔽（SSAO）技术则是一种基于观察空间法线与深度值的实时逼近算法。虽然我们确实可以看出因这种技术的错误结果所导致的一些瑕疵与状况，但对于条件相对有限而仍要计算遮蔽数据的情况来说，这仍是一种实践效果极好的解决方案。

21.4　练习

1．试借助网络研究KD树、四叉树（quadtree）与八叉树（octree）。

2．修改“SSAO”（屏幕空间环境光遮蔽）演示程序，以高斯模糊取代其中的边缘保留模糊。试问，哪种方法更佳呢？

3．能否用计算着色器来实现SSAO呢？如果可以，试给出大致的实现步骤。

4．图21.10所示的是我们不进行自相交检测（self-intersection，参见21.2.2.4节）所生成的SSAO图。试修改“SSAO”例程，去掉其中的自相交检测以欣赏图21.10所示的效果。

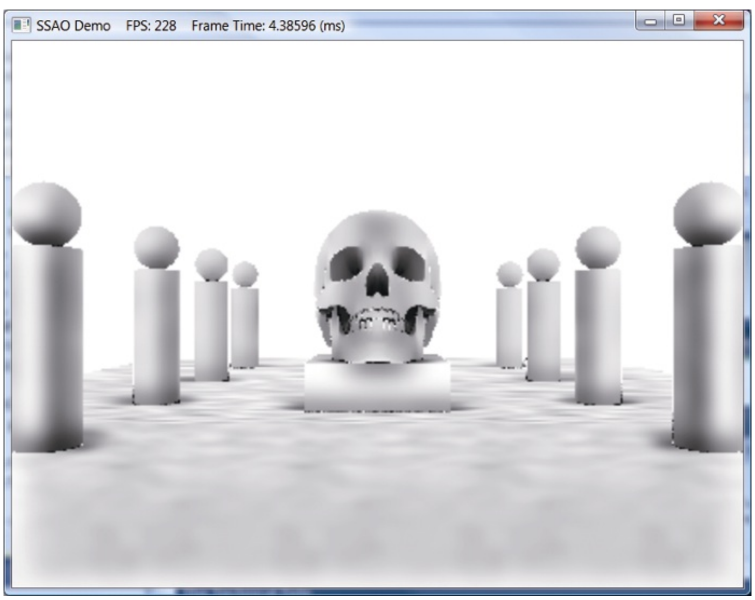


图21.10　到处都是错误的遮蔽效果

[1]　此程序的完整代码详见本书上一版附件d3d11CodeSet3.zip里的Chapter 22 Ambient Occlusion/AmbientOcclusion工程。上述代码也是出自于此。

[2]　该词常用于音频处理方面，此处意为采样较少（仅为完整分辨率的1/4，采样间隔大），也就是采样频率低的特效。

第22章　四元数[1]

我们在第1章中曾介绍过一种名为向量（矢量）的数学对象。还特意学习过由有序实数三元组所构成的3D向量，并用它来定义在几何学中十分有用的向量运算。此外，亦有如第2章中讲解的矩阵——这是一种以实数排列而成的矩形表，由它们定义的多种运算在几何学上也必不可少。比如说，我们曾验证矩阵可以表示线性变换与仿射变换，并将矩阵乘法用于多种变换的复合。在本章中，我们将考察一种名为四元数（quaternion）的数学对象。在此，我们将看到单位四元数（unit quaternion）可用于表示3D旋转，并具有便利的插值性质。对于希望在四元数（及其旋转变换）方面有更深入理解的读者，我们诚挚地为您推荐此主题相关的书籍[Kuipers99]。

学习目标：

1．温习复数知识并回顾如何用复数的乘法运算来表示平面内的旋转操作。

2．理解四元数及其基本运算的定义。

3．探索如何用一组单位四元数来表示一系列3D旋转操作。

4．研究如何在各种不同的旋转表示法之间进行转换。

5．学习如何在单位四元数之间进行插值，并理解这在几何学上等同于在3D方向之间进行插值。

6．熟悉DirectXMath库中与四元数相关的函数与类。

22.1 复数回顾

四元数（quaternion）可被视为复数的推广，这便是我们在开始学习四元数之前先回顾复数的原因。特别地，我们在本节的主要目标是证明复数$\boldsymbol{p}$（可看作一个2D向量或2D点）与单位复数相乘的结果，即为在几何学上对$\boldsymbol{p}$进行相应的旋转操作。而在22.3节中，我们还将证明：一个特定的四元数与一个单位四元数（unit quaternion）乘积的结果，就相当于在几何学上对一个向量或点$\boldsymbol{p}$执行对应的3D旋转操作。

22.1.1 定义

从不同的角度去看，复数的解释方式可谓五花八门。由于一见到它就会使我们立即联想到2D点或2D向量，那么我们不妨就以这样的方式来介绍它吧。

有序实数对$\boldsymbol{z}=(a,b)$表示一个复数。其第一个分量名为实部（real part），第二个分量则称为虚部（imaginary part）。据此，复数相等、加法运算、减法运算、乘法运算以及除法运算的定义依次为：

1. $(a,\ b)=(c,\ d)$，当且仅当$a=c$且$b=d$。

2. $(a,\ b)\pm(c,\ d)=(a\pm c,\ b\pm d)$。

3. $(a,\ b)(c,\ d)=(ac-bd,\ ad+bc)$。

4．如果$(c,d)\neq(0,0)$，那么$\dfrac{(a,b)}{(c,d)}=\left(\dfrac{ac+bd}{c^2+d^2},\dfrac{bc-ad}{c^2+d^2}\right)$。

我们可以轻易地证明出实数常见的算术性质（例如交换律、结合律以及分配律）也适用于复数运算。这一具体应用可参见本章练习1。

形如$(x,0)$的复数通常直接以实数x来表示，并记作$x=(x,0)$。这样一来，任何实数都可看作虚部为0的复数。通过观察可以发现，实数与复数的乘积可由$x(a,b)=(x,0)(a,b)=(xa,xb)=(a,b)(x,0)=(a,b)x$给出。对此，读者有没有联想起标量和向量之间的乘法运算呢？

定义虚数单位（imaginary unit）$i=(0,1)$。根据复数的乘法定义，可得到$i^2=(0,\ 1)(0,\ 1)=(-1,\ 0)=-1$，这表明$i=\sqrt{-1}$。也就是说，$i$为方程$x^2=-1$的解。

复数$\boldsymbol{z}=(a,b)$的共轭复数（complex conjugate）记作$\bar{\boldsymbol{z}}$，并表示为$\bar{\boldsymbol{z}}=(a,-b)$。复数除法公式的一种简便助记方法是为分子和分母同时乘以分母的共轭复数。如此一来，分母部分就成为了一个实数：

$$\frac{(a,b)}{(c,d)}=\frac{(a,b)}{(c,d)}\frac{(c,-d)}{(c,-d)}=\frac{(ac+bd,bc-ad)}{c^2+d^2}=\left(\frac{ac+bd}{c^2+d^2},\frac{bc-ad}{c^2+d^2}\right)$$

接下来，我们证明复数(a,b)可以写作$a+\boldsymbol{i}b$的形式。假设已知$a=(a,0),b=(b,0),i=(0,1)$，则有

$$a+ib=(a,0)+(0,1)(b,0)=(a,0)+(0,b)=(a,b)$$

通过$(a+ib)$的形式，我们可以将复数的加、减、乘、除四则运算重新定义为：

1. $(a+ib)\pm(c+id)=(a\pm c)+i(b\pm d)$。

2. $(a+ib)(c+id)=(ac-bd)+i(ad+bc)$。

3. 如果$(c,d)\neq(0,0)$，那么$\frac{a+ib}{c+id}=\frac{ac+bd}{c^2+d^2}+i\frac{bc-ad}{c^2+d^2}$。

另外，在这种形式下，$\boldsymbol{z}=a+ib$的共轭复数则为$\bar{\boldsymbol{z}}=a-ib$。

22.1.2 复数的几何意义

复数的有序实数对形式$a+ib=(a,b)$，使我们自然而然地将复数与几何学中复平面内的2D点或2D向量联系到一起。事实上，复数加法运算的定义与向量加法的定义一致，如图22.1所示。在下一节中，我们将给出复数乘法运算的几何意义。

复数$a+ib$的绝对值（absolute value）或称为模（magnitude）可由对应向量的长度来予以表示（见图22.2），可通过下式来求得此值：

$$|a+ib|=\sqrt{a^2+b^2}$$

如果一个复数的模为1，我们就称它为单位复数（unit complex number）。

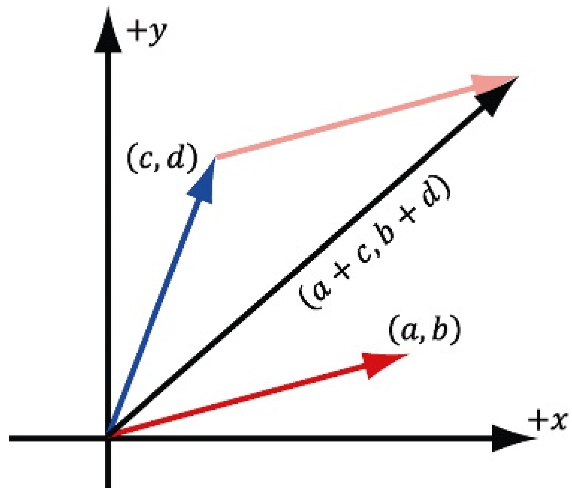


图22.1　复数加法不禁让人联想起平面中的向量加法

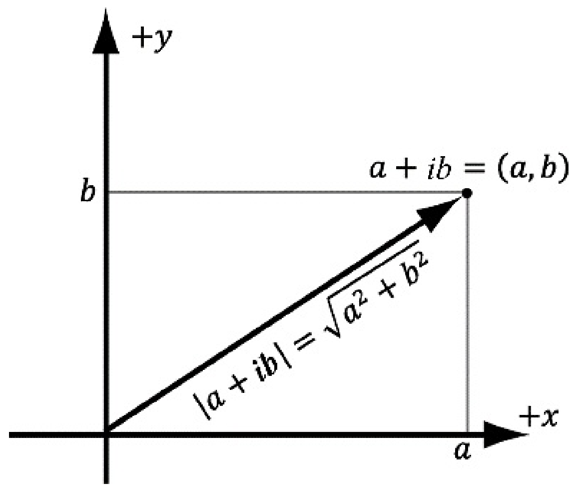


图22.2　复数的模

22.1.3　极坐标表示法与旋转操作

由于可以将复数视作2D复平面内的点或向量，因此我们就能将它们的分量用极坐标（polar coordinate）来表示，如图22.3所示。

$$r = |a + ib|$$

$$a + ib = r\cos\boldsymbol{\theta} + ir\sin\boldsymbol{\theta} = r(\cos\boldsymbol{\theta} + i\sin\boldsymbol{\theta})$$

等式右侧称为复数$a+ib$的极坐标表示（polar representation）。

我们下面以极坐标的形式来表示两个复数的乘法运算。设$\boldsymbol{z}_1=r_1(\cos\boldsymbol{\theta}_1+i\sin\boldsymbol{\theta}_1)$以及$\boldsymbol{z}_2=r_2(\cos\boldsymbol{\theta}_2+i\sin\boldsymbol{\theta}_2)$。则有

$$\begin{aligned}\boldsymbol{z}_1\boldsymbol{z}_2&=r_1r_2(\cos\boldsymbol{\theta}_1\cos\boldsymbol{\theta}_2-\sin\boldsymbol{\theta}_1\sin\boldsymbol{\theta}_2+i(\cos\boldsymbol{\theta}_1\sin\boldsymbol{\theta}_2+\sin\boldsymbol{\theta}_1\cos\boldsymbol{\theta}_2))\\&=r_1r_2(\cos(\boldsymbol{\theta}_1+\boldsymbol{\theta}_2)+i\sin(\boldsymbol{\theta}_1+\boldsymbol{\theta}_2))\end{aligned}$$

推导过程中我们运用了三角恒等式

$$\begin{aligned}\sin(\alpha+\boldsymbol{\beta})&=\sin\alpha\cos\boldsymbol{\beta}+\cos\alpha\sin\boldsymbol{\beta}\\\cos(\alpha+\boldsymbol{\beta})&=\cos\alpha\cos\boldsymbol{\beta}-\sin\alpha\sin\boldsymbol{\beta}\end{aligned}$$

因此，从几何学角度上来看，乘积$\boldsymbol{z}_1\boldsymbol{z}_2$所得到的复数可表示为由模为$r_1r_2$且与实轴夹角为$\boldsymbol{\theta}_1+\boldsymbol{\theta}_2$的向量。特别地，如果$r_2=1$，则$z_1z_2=r_1\left(\cos\left(\boldsymbol{\theta}_1+\boldsymbol{\theta}_2\right)+i\sin\left(\boldsymbol{\theta}_1+\boldsymbol{\theta}_2\right)\right)$，在几何学上即表示将$z_1$按$\boldsymbol{\theta}_2$进行旋转，如图22.4所示。因此，将复数$\boldsymbol{z}_1$（把它视为一个**2D**点或**2D**向量）与单位复数$\boldsymbol{z}_2$相乘就相当于对$\boldsymbol{z}_1$进行旋转操作。

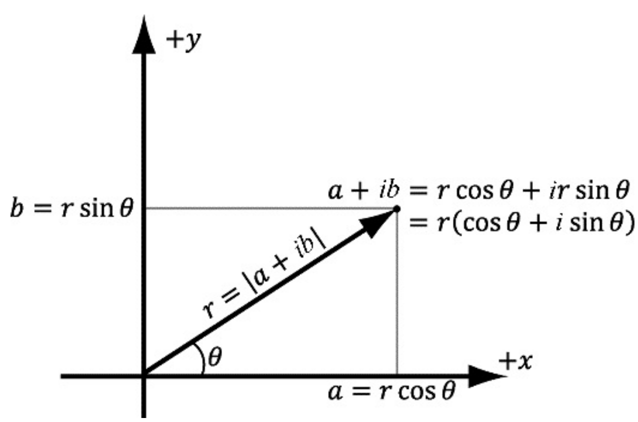


图22.3　一个复数的极坐标表示

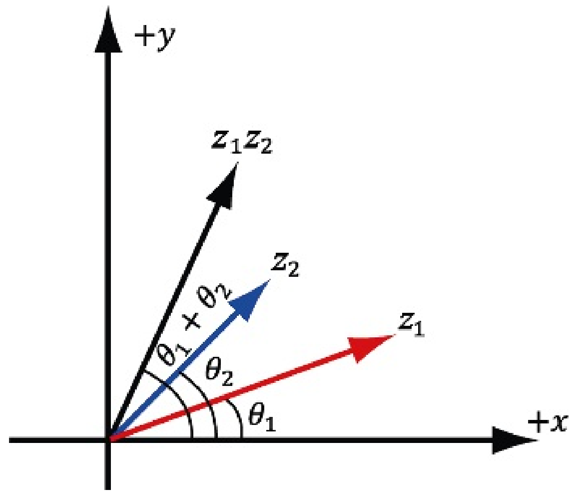


图22.4　$\boldsymbol{z}_1 = r_1(\cos\boldsymbol{\theta}_1 + i\sin\boldsymbol{\theta}_1)$，$\boldsymbol{z}_2 = (\cos\boldsymbol{\theta}_2 + i\sin\boldsymbol{\theta}_2)$。乘积$\boldsymbol{z}_1\boldsymbol{z}_2$就相当于将$\boldsymbol{z}_1$按角$\boldsymbol{\theta}_2$进行旋转

22.2　四元数代数

22.2.1　定义与基本运算

有序实数四元组$\boldsymbol{q}=(x,\ y,\ z,\ w)=(q_1,\ q_2,\ q_3,\ q_4)$即为一个四元数。通常可将它简记作$\boldsymbol{q}=(\boldsymbol{u},\ w)=(x,\ y,\ z,\ w)$，称$\boldsymbol{u}=(x,\ y,\ z)$为虚部向量（imaginary vector part，虚向量部分），而w为实部。据此，四元数相等、加法运算、减法运算、乘法运算的定义如下。

1. $(\boldsymbol{u},\ a)=(\boldsymbol{v},\ b)$，当且仅当$\boldsymbol{u}=\boldsymbol{v}$且$a=b$。

2. $(\boldsymbol{u},\ a)\pm(\boldsymbol{v},\ b)=(\boldsymbol{u}\pm\boldsymbol{v},\ a\pm b)$。

3. $(\boldsymbol{u},\ a)(\boldsymbol{v},\ b)=(a\boldsymbol{v}+b\boldsymbol{u}+\boldsymbol{u}\times\boldsymbol{v},\ ab-\boldsymbol{u}\cdot\boldsymbol{v})$。

尽管四元数乘法运算的定义看起来似乎有点“奇葩”，但这确实是不争的事实。虽然如此，我们却可以根据这些定义来推导它的其他定义形式，而这推导的结果会更便于使用。下面我们就来将四元数之间的乘法定义为矩阵的乘法。

设$\boldsymbol{p}=(\boldsymbol{u},\ p_4)=(p_1,\ p_2,\ p_3,\ p_4)$以及$\boldsymbol{q}=(\boldsymbol{v},\ q_4)=(q_1,\ q_2,\ q_3,\ q_4)$。那么$\boldsymbol{u}\times\boldsymbol{v}=(p_2q_3-p_3q_2,\ p_3q_1-p_1q_3, p_1q_2-p_2q_1)$[2]，且$\boldsymbol{u}\cdot\boldsymbol{v}=p_1q_1+p_2q_2+p_3q_3$。现在，四元数乘积$\boldsymbol{r}=\boldsymbol{pq}$就能够表示为以下的分量形式：

$$
\begin{aligned}
r_1 &= p_4q_1 + q_4p_1 + p_2q_3 - p_3q_2 = q_1p_4 - q_2p_3 + q_3p_2 + q_4p_1 \\
r_2 &= p_4q_2 + q_4p_2 + p_3q_1 - p_1q_3 = q_1p_3 - q_2p_4 + q_3p_1 + q_4p_2 \\
r_3 &= p_4q_3 + q_4p_3 + p_1q_2 - p_2q_1 = -q_1p_2 + q_2p_1 + q_3p_4 + q_4p_3 \\
r_4 &= p_4q_4 - p_1q_1 - p_2q_2 - p_3q_3 = -q_1p_1 - q_2p_2 - q_3p_3 + q_4p_4
\end{aligned}
$$

这可以写作矩阵的乘积：

$$
\boldsymbol{pq} = \begin{bmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix}
$$

注意 Note

如果读者更偏爱使用行向量与矩阵的乘积形式，那么就可简单地取其转置矩阵：

$$
\left(\begin{bmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} \right)^{\mathrm{T}} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{bmatrix}^{\mathrm{T}}
$$

22.2.2　特殊乘积

设$\boldsymbol{i} = (1,0,0,0)$，$\boldsymbol{j} = (0,1,0,0)$，$\boldsymbol{k} = (0,0,1,0)$都为四元数。运用这三个数，便能得到一些特殊的乘积，其中一部分会使人联想到叉积计算：

$$
\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = \boldsymbol{ijk} = -1
$$

$$\boldsymbol{ij} = \boldsymbol{k} = -\boldsymbol{ji}$$

$$\boldsymbol{jk} = \boldsymbol{i} = -\boldsymbol{kj}$$

$$\boldsymbol{ki} = \boldsymbol{j} = -\boldsymbol{ik}$$

这些等式可由之前推算的四元数乘法定义直接推导出。例如，

$$\boldsymbol{ij} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \boldsymbol{k}$$

22.2.3　性质

四元数乘法并不满足交换律，例如，我们在22.2.2节中曾证明 $\boldsymbol{ij} = -\boldsymbol{ji}$。但是四元数仍满足结合律，这能从四元数乘法可以写作矩阵乘法这一点看出：因为矩阵乘法就满足结合律。可将四元数 $\boldsymbol{e} = (0, 0, 0, 1)$看作乘法单位元（multiplicative identity）：

$$\boldsymbol{pe} = \boldsymbol{ep} = \begin{bmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

我们还能得出四元数乘法对加法的分配律：$\boldsymbol{p}(\boldsymbol{q} + \boldsymbol{r}) = \boldsymbol{pq} + \boldsymbol{pr}$以及 $(\boldsymbol{q} + \boldsymbol{r})\boldsymbol{p} = \boldsymbol{qp} + \boldsymbol{rp}$。为证明此性质，可将四元数的乘法与加法都写作矩阵形式，另据矩阵乘法对加法的分配律便可证明。

22.2.4　转换

我们将实数、向量（或点）与四元数的关系表达如下。设s为一个实数，且$\boldsymbol{u}=(x,y,z)$为一个向量，那么：

1. $s=(0,0,0,s)$。

2. $\boldsymbol{u}=(x,y,z)=(\boldsymbol{u},0)=(x,y,z,0)$。

换言之，可将实数看作向量虚部为0的四元数，而把向量视为实部为0的四元数。特别是四元数单位元（identity quaternion），1 = (0, 0, 0, 1)。我们把实部为0的四元数称为纯四元数（pure quaternion）。

根据四元数的乘法运算定义可知，一个实数与一个四元数相乘其实就是一种“标量乘法”，而且这种运算还满足交换律：

$$s(p_1,\ p_2,\ p_3,\ p_4)=(0,0,0,s)(p_1,\ p_2,\ p_3,\ p_4)=\begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & s \end{bmatrix}\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}=\begin{bmatrix} sp_1 \\ sp_2 \\ sp_3 \\ sp_4 \end{bmatrix}$$

类似地，

$$(p_1,\ p_2,\ p_3,\ p_4)s=(p_1,\ p_2,\ p_3,\ p_4)(0,0,0,s)=\begin{bmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \\ 0 \\ s \end{bmatrix}=\begin{bmatrix} sp_1 \\ sp_2 \\ sp_3 \\ sp_4 \end{bmatrix}$$

22.2.5 共轭与范数

我们将四元数$\boldsymbol{q} = (q_1, q_2, q_3, q_4) = (\boldsymbol{u}, q_4)$的共轭四元数记作$\boldsymbol{q}*$，它的定义为：

$$\boldsymbol{q}* = -q_1 - q_2 - q_3 + q_4 = (-u, q_4)$$

换句话说，仅是把原四元数的向量虚部变为其相反数而已；这与复数共轭的定义很相似。以下所列的是一些共轭四元数的相关性质：

1. $(\boldsymbol{pq})* = \boldsymbol{q} * \boldsymbol{p}*$。

2. $(\boldsymbol{p} + \boldsymbol{q})* = \boldsymbol{p} * + \boldsymbol{q}*$。

3. $(\boldsymbol{q}*)* = \boldsymbol{q}$。

4. 当$s \in \mathbb{R}$时，$(s\boldsymbol{q})* = s\boldsymbol{q}*$。

5. $\boldsymbol{q} + \boldsymbol{q}* = (\boldsymbol{u}, q_4) + (-\boldsymbol{u}, q_4) = (0, 2q_4) = 2q_4$。

6. $\boldsymbol{qq}* = \boldsymbol{q} * \boldsymbol{q} = q_1^2 + q_2^2 + q_3^2 + q_4^2 = \|\boldsymbol{u}\|^2 + q_4^2$。

特别是，$\boldsymbol{q} + \boldsymbol{q}*$与$\boldsymbol{qq}*$这两种运算的结果为实数。

四元数的范数（norm，或称模，magnitude）被定义为：

$$\|\boldsymbol{q}\| = \sqrt{\boldsymbol{qq}*} = \sqrt{q_1^2 + q_2^2 + q_3^2 + q_4^2} = \sqrt{\|\boldsymbol{u}\|^2 + q_4^2}$$

如果一个四元数的范数为1，我们就称它是一个单位四元数（unit

quaternion）。范数具有下列性质：

1. $\|\boldsymbol{q}*\| = \|\boldsymbol{q}\|$。

2. $\|\boldsymbol{pq}\| = \|\boldsymbol{p}\|\|\boldsymbol{q}\|$。

性质2反映出两个单位四元数的乘积仍是一个单位四元数。而且，如果$\|\boldsymbol{p}\| = 1$，那么$\|\boldsymbol{pq}\| = \|\boldsymbol{q}\|$。

四元数共轭与范数的性质能够直接由其定义推导出。例如：

$$(\boldsymbol{q}*)* = (-\boldsymbol{u}, q_4)* = (\boldsymbol{u}, q_4) = \boldsymbol{q}$$
$$\|\boldsymbol{q}*\| = \|(-\boldsymbol{u}, q_4)\| = \sqrt{\|-\boldsymbol{u}\|^2 + q_4^2} = \sqrt{\|\boldsymbol{u}\|^2 + q_4^2} = \|\boldsymbol{q}\|$$

$$\begin{aligned}\|\boldsymbol{pq}\|^2 &= (\boldsymbol{pq})(\boldsymbol{pq})* \\ &= \boldsymbol{pqq}*\boldsymbol{p}* \\ &= \boldsymbol{p}\|\boldsymbol{q}\|^2\boldsymbol{p}* \\ &= \boldsymbol{pp}*\|\boldsymbol{q}\|^2 \\ &= \|\boldsymbol{p}\|^2\|\boldsymbol{q}\|^2\end{aligned}$$

读者可以尝试对其他的性质进行推导（参见章后练习）。

22.2.6 四元数的逆

与矩阵一样，四元数的乘法运算亦不满足交换律，因此也就无法定义四元数的除法运算（只有在乘法满足交换律时，才可以定义其相应的除法运算，即有$\frac{a}{b} = ab^{-1} = b^{-1}a$）。然而，每个非零四元数（nonzero quaternion，零四元数的每个分量皆为0）都有其相应的逆。设

$\boldsymbol{q}=(q_1,q_2,q_3,q_4)=(\boldsymbol{u},q_4)$为一非零四元数，那么它的逆被记作$\boldsymbol{q}^{-1}$，其定义为：

$$\boldsymbol{q}^{-1}=\frac{\boldsymbol{q}*}{||\boldsymbol{q}||^2}$$

四元数之逆的检验十分方便，对此，我们有：

$$\boldsymbol{q}\boldsymbol{q}^{-1}=\frac{\boldsymbol{q}\boldsymbol{q}*}{||\boldsymbol{q}||^2}=\frac{||\boldsymbol{q}||^2}{||\boldsymbol{q}||^2}=1=(0,0,0,1)$$
$$\boldsymbol{q}^{-1}\boldsymbol{q}=\frac{\boldsymbol{q}*\boldsymbol{q}}{||\boldsymbol{q}||^2}=\frac{||\boldsymbol{q}||^2}{||\boldsymbol{q}||^2}=1=(0,0,0,1)$$

可以发现，如果$\boldsymbol{q}$是一个单位四元数，那么$||\boldsymbol{q}||^2=1$，因此$\boldsymbol{q}^{-1}=\boldsymbol{q}*$。

四元数的逆具有下列性质：

1. $\left(\boldsymbol{q}^{-1}\right)^{-1}=\boldsymbol{q}$。

2. $(\boldsymbol{p}\boldsymbol{q})^{-1}=\boldsymbol{q}^{-1}\boldsymbol{p}^{-1}$。

22.2.7 极坐标表示法

如果$\boldsymbol{q}=(q_1,q_2,q_3,q_4)=(\boldsymbol{u},q_4)$为一个单位四元数，那么

$$||\boldsymbol{q}||^2=||\boldsymbol{u}||^2+q_4^2=1$$

这说明$q_4^2\leqslant 1\Leftrightarrow |q_4|\leqslant 1\Leftrightarrow -1\leqslant q_4\leqslant 1$。图22.5展示了存在这样一个角度$\boldsymbol{\theta}\in[0,\pi]$使$q_4=\cos\boldsymbol{\theta}$。根据三角恒等式$\sin^2\boldsymbol{\theta}+\cos^2\boldsymbol{\theta}=1$，有

$$\sin^2\boldsymbol{\theta} = 1 - \cos^2\boldsymbol{\theta} = 1 - q_4^2 = ||\boldsymbol{u}||^2$$

这就意味着

$$||\boldsymbol{u}|| = |\sin\boldsymbol{\theta}| = \sin\boldsymbol{\theta}\text{，其中}\boldsymbol{\theta} \in [0, \pi]$$

现在，将$\boldsymbol{n}$表示为与向量$\boldsymbol{u}$同方向的单位向量：

$$\boldsymbol{n} = \frac{\boldsymbol{u}}{||\boldsymbol{u}||} = \frac{\boldsymbol{u}}{\sin\boldsymbol{\theta}}$$

因此，$\boldsymbol{u} = \sin\boldsymbol{\theta}\boldsymbol{n}$。据此，我们就能够把单位四元数$\boldsymbol{q} = (u, q_4)$表示为极坐标的形式，其中的$\boldsymbol{n}$为一单位向量：

$$\boldsymbol{q} = (\sin\boldsymbol{\theta}\boldsymbol{n}, \cos\boldsymbol{\theta})\text{，其中}\boldsymbol{\theta} \in [0, \pi]$$

例如，假设我们指定一个四元数$\boldsymbol{q} = \left(0, \frac{1}{2}, 0, \frac{\sqrt{3}}{2}\right)$。为将其转换为极坐标表示法，我们求得$\boldsymbol{\theta} = \arccos\frac{\sqrt{3}}{2} = \frac{\pi}{6}$，$\boldsymbol{n} = \frac{(0, \frac{1}{2}, 0)}{\sin\frac{\pi}{6}} = (0, 1, 0)$。因此，$\boldsymbol{q} = \left(\sin\frac{\pi}{6}(0, 1, 0), \cos\frac{\pi}{6}\right)$。

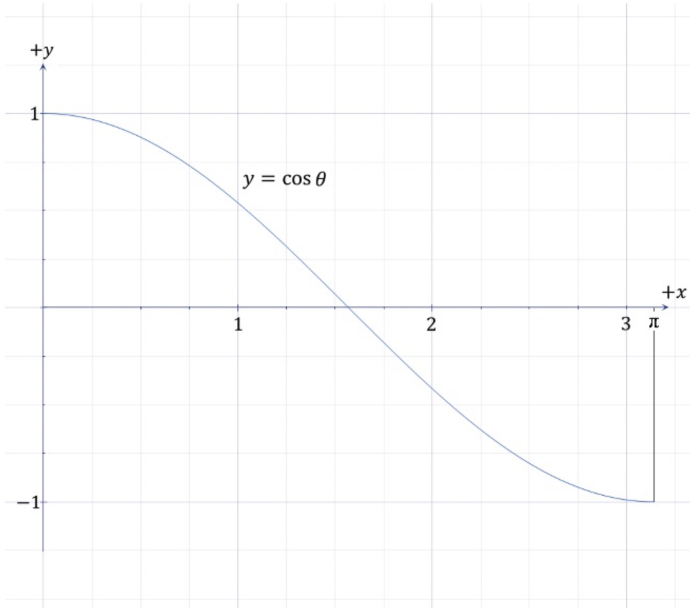


图22.5　对于$y \in [-1,\ 1]$而言，必存在一个角$\boldsymbol{\theta}$使$y = \cos\boldsymbol{\theta}$

注意　Note

$\boldsymbol{\theta} \in [0, \pi]$是将四元数$\boldsymbol{q} = (q_1, q_2, q_3, q_4)$转换至极坐标表示法的限制条件。也就是说，为了把一个唯一的角度与四元数$\boldsymbol{q} = (q_1, q_2, q_3, q_4)$相关联，便需要对其范围进行约束。如果推翻此限制，求取任意角度θ所对应的四元数$\boldsymbol{q} = (\sin\boldsymbol{\theta} n, \cos\boldsymbol{\theta})$，根据$\boldsymbol{q} = (\sin(\boldsymbol{\theta} + 2\pi n)n, \cos(\boldsymbol{\theta} + 2\pi n))$可知，$\boldsymbol{n}$取任意整数均可得到四元数$\boldsymbol{q} = (\sin\boldsymbol{\theta} n, \cos\boldsymbol{\theta})$。所以，如果没有$\boldsymbol{\theta} \in [0, \pi]$这个角度限制，那么四元数的极坐标表示就不是唯一的。

值得注意的是，以$-\boldsymbol{\theta}$代替$\boldsymbol{\theta}$就相当于把四元数的向量虚部变为其相反数：

$$(\boldsymbol{n}\sin(-\boldsymbol{\theta}), \cos(-\boldsymbol{\theta})) = (-\boldsymbol{n}\sin\boldsymbol{\theta}, \cos\boldsymbol{\theta}) = \boldsymbol{p}*$$

在下一节中，我们将认识到$\boldsymbol{n}$表示的是旋转轴，因此我们可以通过上述方法调转旋转轴的方向，使被旋转对象按相反方向进行旋转。

22.3 单位四元数及其旋转操作

22.3.1 旋转算子

设$\boldsymbol{q}=(\boldsymbol{u},w)$为单位四元数，而$\boldsymbol{v}$为一个3D点或3D向量。接下来，我们就可以把$\boldsymbol{v}$视为纯四元数$\boldsymbol{p}=(\boldsymbol{v},0)$。又因为$\boldsymbol{q}$是一个单位四元数，根据其性质，我们有$\boldsymbol{q}^{-1}=\boldsymbol{q}*$。回顾四元数的乘法运算公式可知：

$$(\boldsymbol{m},a)(\boldsymbol{n},b)=(a\boldsymbol{n}+b\boldsymbol{m}+\boldsymbol{m}\times\boldsymbol{n},ab-\boldsymbol{m}\cdot\boldsymbol{n})$$

现在来考虑下列乘积：

$$\begin{aligned}\boldsymbol{qpq}^{-1}&=\boldsymbol{qpq}*\\&=(\boldsymbol{u},w)(\boldsymbol{v},0)(-\boldsymbol{u},w)\\&=(\boldsymbol{u},w)(w\boldsymbol{v}-\boldsymbol{v}\times\boldsymbol{u},\boldsymbol{v}\cdot\boldsymbol{u})\end{aligned}$$

对这个稍有点冗长的公式进行化简，我们将其实部与向量虚部分开处理，用下列符号分别替换式子中的对应项：

$$\begin{aligned}a&=w\\b&=\boldsymbol{v}\cdot\boldsymbol{u}\\\boldsymbol{m}&=\boldsymbol{u}\\\boldsymbol{n}&=w\boldsymbol{v}-\boldsymbol{v}\times\boldsymbol{u}\end{aligned}$$

<u>实部</u>

$$
\begin{aligned}
& ab - \boldsymbol{m} \cdot \boldsymbol{n} \\
& = w(\boldsymbol{v} \cdot \boldsymbol{u}) - \boldsymbol{u} \cdot (w\boldsymbol{v} - \boldsymbol{v} \times \boldsymbol{u}) \\
& = w(\boldsymbol{v} \cdot \boldsymbol{u}) - \boldsymbol{u} \cdot w\boldsymbol{v} + \boldsymbol{u} \cdot (\boldsymbol{v} \times \boldsymbol{u}) \\
& = w(\boldsymbol{v} \cdot \boldsymbol{u}) - w(\boldsymbol{v} \cdot \boldsymbol{u}) + 0 \\
& = 0
\end{aligned}
$$

注意，$\boldsymbol{u} \cdot (\boldsymbol{v} \times \boldsymbol{u}) = 0$是因为根据叉积的定义可知$(\boldsymbol{v} \times \boldsymbol{u})$正交于$\boldsymbol{u}$。

<u>向量虚部</u>

$$
\begin{aligned}
& a\boldsymbol{n} + b\boldsymbol{m} + \boldsymbol{m} \times \boldsymbol{n} \\
& = w(w\boldsymbol{v} - \boldsymbol{v} \times \boldsymbol{u}) + (\boldsymbol{v} \cdot \boldsymbol{u})\boldsymbol{u} + \boldsymbol{u} \times (w\boldsymbol{v} - \boldsymbol{v} \times \boldsymbol{u}) \\
& = w^2\boldsymbol{v} - w\boldsymbol{v} \times \boldsymbol{u} + (\boldsymbol{v} \cdot \boldsymbol{u})\boldsymbol{u} + \boldsymbol{u} \times w\boldsymbol{v} + \boldsymbol{u} \times (\boldsymbol{u} \times \boldsymbol{v}) \\
& = w^2\boldsymbol{v} + \boldsymbol{u} \times w\boldsymbol{v} + (\boldsymbol{v} \cdot \boldsymbol{u})\boldsymbol{u} + \boldsymbol{u} \times w\boldsymbol{v} + \boldsymbol{u} \times (\boldsymbol{u} \times \boldsymbol{v})
\end{aligned}
$$

$$
\begin{aligned}
& = w^2\boldsymbol{v} + 2(\boldsymbol{u} \times w\boldsymbol{v}) + (v \cdot \boldsymbol{u})\boldsymbol{u} + \boldsymbol{u} \times (\boldsymbol{u} \times \boldsymbol{v}) \\
& = w^2\boldsymbol{v} + 2(\boldsymbol{u} \times w\boldsymbol{v}) + (\boldsymbol{v} \cdot \boldsymbol{u})\boldsymbol{u} + (\boldsymbol{u} \cdot \boldsymbol{v})\boldsymbol{u} - (\boldsymbol{u} \cdot \boldsymbol{u})\boldsymbol{v} \\
& = (w^2 - \boldsymbol{u} \cdot \boldsymbol{u})\boldsymbol{v} + 2w(\boldsymbol{u} \times \boldsymbol{v}) + 2(\boldsymbol{u} \cdot \boldsymbol{v})\boldsymbol{u} \\
& = (w^2 - \boldsymbol{u} \cdot \boldsymbol{u})\boldsymbol{v} + 2(\boldsymbol{u} \cdot \boldsymbol{v})\boldsymbol{u} + 2w(\boldsymbol{u} \times \boldsymbol{v})
\end{aligned}
$$

在这个过程中，我们利用了向量三重积
$\boldsymbol{a} \times (\boldsymbol{b} \times \boldsymbol{c}) = (\boldsymbol{a} \cdot \boldsymbol{c})\boldsymbol{b} - (\boldsymbol{a} \cdot \boldsymbol{b})\boldsymbol{c}$来变换$\boldsymbol{u} \times (\boldsymbol{u} \times \boldsymbol{v})$[3]。

这样，我们就证明出：

$$
\boldsymbol{qpq}* = ((w^2 - \boldsymbol{u} \cdot \boldsymbol{u})\boldsymbol{v} + 2(\boldsymbol{u} \cdot \boldsymbol{v})\boldsymbol{u} + 2w(\boldsymbol{u} \times \boldsymbol{v}), 0) \qquad (22.1)
$$

可以看出该结果是一个向量或一个点，因为其实部为0（如果要利用该算子对一个向量或点进行旋转，那么满足此条件是有必要的——其求值结果必须是一个向量或点）。因此，我们在随后的公式中将省略四元数的实部。

由于$\boldsymbol{q}$为单位四元数，便可将它写作

$$\boldsymbol{q}=(\sin\boldsymbol{\theta}\boldsymbol{n},\cos\boldsymbol{\theta})\text{，其中}\|\boldsymbol{n}\|=1\text{且}\boldsymbol{\theta}\in[0,\pi]$$

再把它代入式（22.1）中：

$$\begin{aligned}\boldsymbol{q}\boldsymbol{p}\boldsymbol{q}*&=(\cos^2\boldsymbol{\theta}-\sin^2\boldsymbol{\theta})\boldsymbol{v}+2(\sin\boldsymbol{\theta}\boldsymbol{n}\cdot\boldsymbol{v})\sin\boldsymbol{\theta}\boldsymbol{n}+2\cos\boldsymbol{\theta}(\sin\boldsymbol{\theta}\boldsymbol{n}\times\boldsymbol{v})\\&=(\cos^2\boldsymbol{\theta}-\sin^2\boldsymbol{\theta})\boldsymbol{v}+2\sin^2\boldsymbol{\theta}(\boldsymbol{n}\cdot\boldsymbol{v})\boldsymbol{n}+2\cos\boldsymbol{\theta}\sin\boldsymbol{\theta}(\boldsymbol{n}\times\boldsymbol{v})\end{aligned}$$

运用三角恒等式对它进一步化简：

$$\begin{aligned}&\cos^2\boldsymbol{\theta}-\sin^2\boldsymbol{\theta}=\cos(2\boldsymbol{\theta})\\&2\cos\boldsymbol{\theta}\sin\boldsymbol{\theta}=\sin(2\boldsymbol{\theta})\end{aligned}$$

$$\cos(2\boldsymbol{\theta})=1-2\sin^2\boldsymbol{\theta}$$

$$\begin{aligned}\boldsymbol{q}\boldsymbol{p}\boldsymbol{q}*&=(\cos^2\boldsymbol{\theta}-\sin^2\boldsymbol{\theta})\boldsymbol{v}+2\sin^2\boldsymbol{\theta}(\boldsymbol{n}\cdot\boldsymbol{v})\boldsymbol{n}+2\cos\boldsymbol{\theta}\sin\boldsymbol{\theta}\\&=\cos(2\boldsymbol{\theta})\boldsymbol{v}+(1-\cos(2\boldsymbol{\theta}))(\boldsymbol{n}\cdot\boldsymbol{v})\boldsymbol{n}+\sin(2\boldsymbol{\theta})(\boldsymbol{n}\times\boldsymbol{v})\end{aligned}\qquad（22.2）$$

将式（22.2）与以旋转轴及旋转角表示的旋转式（3.5），进行比对可以发现：前者其实就是旋转公式$\boldsymbol{R}_{\mathbf{n}}(\boldsymbol{v})$。换言之，式（22.2）令向量（或点）$\boldsymbol{v}$绕轴$\boldsymbol{n}$按角$2\theta$进行旋转。

$$\boldsymbol{R}_n(\boldsymbol{v})=\cos\boldsymbol{\theta}\boldsymbol{v}+(1-\cos\boldsymbol{\theta})(\boldsymbol{n}\cdot\boldsymbol{v})\boldsymbol{n}+\sin\boldsymbol{\theta}(\boldsymbol{n}\times\boldsymbol{v})$$

因此，我们定义四元数的旋转算子（rotation operator）为：

$$\begin{aligned}R_{\mathrm{q}}(\boldsymbol{v})&=\boldsymbol{q}\boldsymbol{v}\boldsymbol{q}^{-1}\\&=\boldsymbol{q}\boldsymbol{v}\boldsymbol{q}*\\&=\cos(2\boldsymbol{\theta})\boldsymbol{v}+(1-\cos(2\boldsymbol{\theta}))(\boldsymbol{n}\cdot\boldsymbol{v})\boldsymbol{n}+\sin(2\boldsymbol{\theta})(\boldsymbol{n}\times\boldsymbol{v})\end{aligned}\qquad（22.3）$$

我们方才证明了四元数旋转算子$R_{\mathrm{q}}(\boldsymbol{v})=\boldsymbol{q}\boldsymbol{v}\boldsymbol{q}^{-1}$将向量（或点）$\boldsymbol{v}$绕

轴$\boldsymbol{n}$旋转角2θ。

所以，若给定一个旋转轴$\boldsymbol{n}$以及旋转角θ，我们就能通过下式构建出相应的旋转四元数：

$$\boldsymbol{q} = \left(\sin\left(\frac{\boldsymbol{\theta}}{2}\right)\boldsymbol{n}, \cos\left(\frac{\boldsymbol{\theta}}{2}\right)\right)$$

接着，再运用公式$R_q(\boldsymbol{v})$即可得到对应旋转算子。我们在这里将旋转角度除以2是为了与公式中的$2\boldsymbol{\theta}$相抵消，这是因为我们希望旋转的角度为$\boldsymbol{\theta}$，而非$2\boldsymbol{\theta}$。

22.3.2　将四元数旋转算子转换为矩阵形式

设$\boldsymbol{q} = (\boldsymbol{u}, w) = (q_1, q_2, q_3, q_4)$为单位四元数。根据式（22.1）可知：

$$\boldsymbol{r} = R_{\mathrm{q}}(\boldsymbol{v}) = \boldsymbol{qvq}* = (w^2 - \boldsymbol{u}\cdot\boldsymbol{u})\boldsymbol{v} + 2(\boldsymbol{u}\cdot\boldsymbol{v})\boldsymbol{u} + 2w(\boldsymbol{u}\times\boldsymbol{v})$$

注意到由$q_1^2 + q_2^2 + q_3^2 + q_4^2 = 1$（单位四元数的范数为1）可得$q_4^2 - 1 = -q_1^2 - q_2^2 - q_3^2$，因此

$$\begin{aligned}(w^2 - \boldsymbol{u}\cdot\boldsymbol{u})\boldsymbol{v} &= (q_4^2 - q_1^2 - q_2^2 - q_3^2)\boldsymbol{v}\\ &= (2q_4^2 - 1)\boldsymbol{v}\end{aligned}$$

$R_q(\boldsymbol{v})$中的三项可以分别写作矩阵形式：

$$(w^2 - \boldsymbol{u}\cdot\boldsymbol{u})\boldsymbol{v} = [v_x \quad v_y \quad v_z]\begin{bmatrix}2q_4^2-1 & 0 & 0\\ 0 & 2q_4^2-1 & 0\\ 0 & 0 & 2q_4^2-1\end{bmatrix}$$

$$2(\boldsymbol{u}\cdot\boldsymbol{v})\boldsymbol{u}=[v_x \quad v_y \quad v_z]\begin{bmatrix}2q_1^2 & 2q_1q_2 & 2q_1q_3\\ 2q_1q_2 & 2q_2^2 & 2q_2q_3\\ 2q_1q_3 & 2q_2q_3 & 2q_3^2\end{bmatrix}$$

$$2w(\boldsymbol{u}\times\boldsymbol{v})=[v_x \quad v_y \quad v_z]\begin{bmatrix}0 & 2q_4q_3 & -2q_4q_2\\ -2q_4q_3 & 0 & 2q_1q_1\\ 2q_4q_2 & -2q_4q_1 & 0\end{bmatrix}$$

对这三项求和，得到：

$$R_{\mathrm{q}}(\boldsymbol{v})=\boldsymbol{vQ}=[v_x \quad v_y \quad v_z]\begin{bmatrix}2q_1^2+2q_4^2-1 & 2q_1q_2+2q_3q_4 & 2q_1q_3-2q_2q_4\\ 2q_1q_2-2q_3q_4 & 2q_2^2+2q_4^2-1 & 2q_2q_3+2q_1q_4\\ 2q_1q_3+2q_2q_4 & 2q_2q_3-2q_1q_4 & 2q_3^2+2q_4^2-1\end{bmatrix}$$

根据单位四元数$\boldsymbol{q}$的单位长度性质$q_1^2+q_2^2+q_3^2+q_4^2=1$可知：

$$\begin{aligned}2q_1^2+2q_4^2&=2-2q_2^2-2q_3^2\\ 2q_2^2+2q_4^2&=2-2q_1^2-2q_3^2\\ 2q_3^2+2q_4^2&=2-2q_1^2-2q_2^2\end{aligned}$$

因此，我们可以将矩阵方程改写为：

$$R_{\mathrm{q}}(\boldsymbol{v})=\boldsymbol{vQ}=[v_x \quad v_y \quad v_z]\begin{bmatrix}1-2q_2^2-2q_3^2 & 2q_1q_2+2q_3q_4 & 2q_1q_3-2q_2q_4\\ 2q_1q_2-2q_3q_4 & 1-2q_1^2-2q_3^2 & 2q_2q_3+2q_1q_4\\ 2q_1q_3+2q_2q_4 & 2q_2q_3-2q_1q_4 & 1-2q_1^2-2q_2^2\end{bmatrix}$$

（22.4）

注意 Note

许多图形学书籍为了对向量进行变换而采用矩阵与列向量乘积的形式。因此，在某些图形学著作中，我们会看到以矩阵$\boldsymbol{Q}$的转置矩阵所表

示的公式$R_q(\boldsymbol{v})=\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{v}^{\mathrm{T}}$。

22.3.3 将旋转矩阵变换为四元数旋转算子

给出旋转矩阵

$$\boldsymbol{R}=\begin{bmatrix}R_{11} & R_{12} & R_{13}\\ R_{21} & R_{22} & R_{23}\\ R_{31} & R_{32} & R_{33}\end{bmatrix}$$

我们希望能求出四元数$\boldsymbol{q}=(q_1,q_2,q_3,q_4)$，使得我们在用$\boldsymbol{q}$来构建式（22.4）中的矩阵$\boldsymbol{Q}$时得到的是矩阵$\boldsymbol{R}$。因此，这里采取的策略是设：

$$\begin{bmatrix}R_{11} & R_{12} & R_{13}\\ R_{21} & R_{22} & R_{23}\\ R_{31} & R_{32} & R_{33}\end{bmatrix}=\begin{bmatrix}1-2q_2^2-2q_3^2 & 2q_1q_2+2q_3q_4 & 2q_1q_3-2q_2q_4\\ 2q_1q_2-2q_3q_4 & 1-2q_1^2-2q_3^2 & 2q_2q_3+2q_1q_4\\ 2q_1q_3+2q_2q_4 & 2q_2q_3-2q_1q_4 & 1-2q_1^2-2q_2^2\end{bmatrix}$$

再分别解出q_1、q_2、q_3、q_4。注意，由于在开始时给出了矩阵$\boldsymbol{R}$，因此方程左侧的所有元素都是已知的。

首先对位于矩阵主对角线上（左上至右下）的元素求和（此和也称为矩阵的迹，trace）：

$$\begin{aligned}\mathrm{trace}(\boldsymbol{R})&=R_{11}+R_{22}+R_{33}\\ &=1-2\boldsymbol{q}_2^2-2\boldsymbol{q}_3^2+1-2\boldsymbol{q}_1^2-2\boldsymbol{q}_3^2+1-2\boldsymbol{q}_1^2-2\boldsymbol{q}_1^2\\ &=3-4\boldsymbol{q}_1^2-4\boldsymbol{q}_2^2-4\boldsymbol{q}_3^2\\ &=3-4(\boldsymbol{q}_1^2+\boldsymbol{q}_2^2+\boldsymbol{q}_3^2)\\ &=3-4(1-\boldsymbol{q}_4^2)\\ &=-1+4\boldsymbol{q}_4^2\end{aligned}$$

$$\therefore \boldsymbol{q}_4 = \frac{\sqrt{\text{trace}(\boldsymbol{R}) + 1}}{2}$$

现在来组合矩阵主对角线两侧的对称元素以求取q_1、q_2、q_3（因为在上述计算的过程中消去了这几项）：

$$\begin{aligned} & R_{23} - R_{32} = 2q_2q_3 + 2q_1q_4 - 2q_2q_3 + 2q_1q_4 \\ & = 4q_1q_4 \\ & \qquad \therefore\ q_1 \ = \frac{R_{23} - R_{32}}{4q_4} \end{aligned}$$

$$\begin{aligned} & R_{31} - R_{13} = 2q_1q_3 + 2q_2q_4 - 2q_1q_3 + 2q_2q_4 \\ & = 4q_2q_4 \\ & \qquad \therefore\ q_2 \ = \frac{R_{31} - R_{13}}{4q_4} \end{aligned}$$

$$\begin{aligned} & R_{12} - R_{21} = 2q_1q_2 + 2q_3q_4 - 2q_1q_2 + 2q_3q_4 \\ & = 4q_3q_4 \\ & \qquad \therefore\ q_3 \ = \frac{R_{12} - R_{21}}{4q_4} \end{aligned}$$

如果$q_4 = 0$，那么这些公式皆无定义。若发生这种情况，应找到矩阵$\boldsymbol{R}$主对角线上的最大元素来计算被除数，并将其他矩阵元素以另一种方式进行组合。假设R_{11}是矩阵主对角线上的最大元素：

$$\begin{aligned} & R_{11} - R_{22} - R_{33} = 1 - 2q_2^2 - 2q_3^2 - 1 + 2q_1^2 + 2q_3^2 - 1 + 2q_1^2 + 2q_2^2 \\ & = -1 + 4q_1^2 \\ & \qquad \therefore\ q_1 \ = \frac{\sqrt{R_{11} - R_{22} - R_{33} + 1}}{2} \end{aligned}$$

$$\begin{aligned} & R_{11} - R_{12} + R_{21} = 2q_1q_2 + 2q_3q_4 + 2q_1q_2 - 2q_3q_4 \\ & = 4q_1q_2 \\ & \qquad \therefore\ q_2 \ = \frac{R_{12} + R_{21}}{4q_1} \end{aligned}$$

$$R_{13} - R_{31} = 2q_1q_3 - 2q_2q_4 + 2q_2q_4$$
$$= 4q_1q_3$$
$$\therefore\ q_3 = \frac{R_{13} + R_{31}}{4q_1}$$

$$R_{23} - R_{32} = 2q_2q_3 + 2q_1q_4 - 2q_2q_3 + 2q_1q_4$$
$$= 4q_1q_4$$
$$\therefore\ q_4 = \frac{R_{23} + R_{32}}{4q_1}$$

如果主对角线上的最大值为R_{22}或R_{33}，则进行类似的处理。

22.3.4　复合

假设$\boldsymbol{p}$与$\boldsymbol{q}$分别为旋转算子R_p与R_q中所用的相应单位四元数。设$\boldsymbol{v}' = R_p(\boldsymbol{v})$，那么这两个旋转算子的复合过程为：

$$R_q(R_p(\boldsymbol{v})) = R_q(\boldsymbol{v}') = \boldsymbol{q}\boldsymbol{v}'\boldsymbol{q}^{-1} = \boldsymbol{q}(\boldsymbol{p}\boldsymbol{v}\boldsymbol{p}^{-1})\boldsymbol{q}^{-1} = (\boldsymbol{q}\boldsymbol{p})\boldsymbol{v}(\boldsymbol{p}^{-1}\boldsymbol{q}^{-1}) = (\boldsymbol{q}\boldsymbol{p})\boldsymbol{v}(\boldsymbol{q}\boldsymbol{p})^{-1}$$

由于$\boldsymbol{p}$与$\boldsymbol{q}$都为单位四元数，也就是说$\|\boldsymbol{p}\boldsymbol{q}\| = \|\boldsymbol{p}\|\|\boldsymbol{q}\| = 1$，所以乘积$\boldsymbol{p}\boldsymbol{q}$亦为单位四元数。因此，四元数乘积$\boldsymbol{p}\boldsymbol{q}$也表示着一种旋转操作。换句话说，净旋转（net rotation）可表示为旋转算子的复合$R_q(R_p(\boldsymbol{v}))$。

22.4　四元数插值

由于四元数即实数四元组，所以就能将其视作几何学上的4D向量。单位四元数则是位于4D单位球面上的4D单位向量。利用叉积（此运算定义仅用于3D向量）以外的运算规则，我们就能将向量的数学运算推广到四维空间乃至n维空间。尤其是适用于四元数的点积运算：设四元数$\boldsymbol{p}=(\boldsymbol{u},s)$与$\boldsymbol{q}=(\boldsymbol{v},t)$，那么

$$\boldsymbol{p}\cdot\boldsymbol{q}=\boldsymbol{u}\cdot\boldsymbol{v}+st=\|\boldsymbol{p}\|\,\|\boldsymbol{q}\|\cos\boldsymbol{\theta}$$

其中，$\boldsymbol{\theta}$为这两个四元数之间的夹角。如果四元数$\boldsymbol{p}$与$\boldsymbol{q}$皆为单位长度，那么$\boldsymbol{p}\cdot\boldsymbol{q}=\cos\boldsymbol{\theta}$。这就是说，点积使我们可以描述出两个四元数之间的夹角，作为它们在单位球面上彼此远近程度的度量手段。

考虑到3D旋转动画方面上的需求，我们希望在两个不同的方向之间进行插值，求取中间的变化过程。为了对四元数进行插值，我们就联想到对单位球面上的弧进行插值，这样一来，所得到的插值四元数依然是单位四元数。为了推导出这样一个插值公式，我们考虑图22.6所示的情景，在这里，我们希望在$\boldsymbol{a}$到$\boldsymbol{b}$之间的角$t\boldsymbol{\theta}$方向进行插值。此时，我们的目标是求出权重c_1与c_2使$\boldsymbol{p}=c_1\boldsymbol{a}+c_2\boldsymbol{b}$，其中$\|\boldsymbol{p}\|=\|\boldsymbol{a}\|=\|\boldsymbol{b}\|$。现为两个未知权值列出下述等式：

$$\begin{aligned}
&\boldsymbol{a}\cdot\boldsymbol{p}=c_1\boldsymbol{a}\cdot\boldsymbol{a}+c_2\boldsymbol{a}\cdot\boldsymbol{b}\\
&\cos(t\boldsymbol{\theta})=c_1+c_2\cos(\boldsymbol{\theta})\\
&\boldsymbol{p}\cdot\boldsymbol{b}=c_1\boldsymbol{a}\cdot\boldsymbol{b}+c_2\boldsymbol{b}\cdot\boldsymbol{b}\\
&\cos((1-t)\boldsymbol{\theta})=c_1\cos(\boldsymbol{\theta})+c_2
\end{aligned}$$

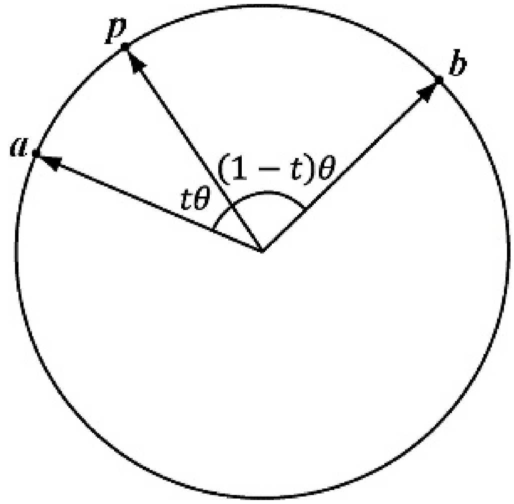


图22.6　沿4D单位球面在由$\boldsymbol{a}$到$\boldsymbol{b}$的角$t\theta$处进行插值。$\boldsymbol{a}$与$\boldsymbol{b}$之间的夹角为$\boldsymbol{\theta}$，$\boldsymbol{a}$与$\boldsymbol{p}$之间的夹角为$t\boldsymbol{\theta}$，而$\boldsymbol{p}$与$\boldsymbol{b}$之间的夹角则为$(1-t)\boldsymbol{\theta}$

用矩阵方程来表示为：

$$\begin{bmatrix} 1 & \cos(\boldsymbol{\theta}) \\ \cos(\boldsymbol{\theta}) & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \cos(t\boldsymbol{\theta}) \\ \cos((1-t)\boldsymbol{\theta}) \end{bmatrix}$$

考虑矩阵方程$\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$，其中的矩阵$\boldsymbol{A}$是可逆的。根据克莱姆法则（Cramer’s Rule）可知，$x_i = \det \boldsymbol{A}_i / \det \boldsymbol{A}$，用列向量$\boldsymbol{b}$替换矩阵$\boldsymbol{A}$中第$i$列的列向量即可得到$\boldsymbol{A}_i$。因此：

$$c_1 = \frac{\det \begin{bmatrix} \cos(t\boldsymbol{\theta}) & \cos(\boldsymbol{\theta}) \\ \cos((1-t)\boldsymbol{\theta}) & 1 \end{bmatrix}}{\det \begin{bmatrix} 1 & \cos(\boldsymbol{\theta}) \\ \cos(\boldsymbol{\theta}) & 1 \end{bmatrix}} = \frac{\cos(t\boldsymbol{\theta}) - \cos(\boldsymbol{\theta})\cos((1-t)\boldsymbol{\theta})}{1-\cos^2(\boldsymbol{\theta})}$$

$$c_2 = \frac{\det \begin{bmatrix} 1 & \cos(t\boldsymbol{\theta}) \\ \cos(\boldsymbol{\theta}) & \cos((1-t)\boldsymbol{\theta}) \end{bmatrix}}{\det \begin{bmatrix} 1 & \cos(\boldsymbol{\theta}) \\ \cos(\boldsymbol{\theta}) & 1 \end{bmatrix}} = \frac{\cos((1-t)\boldsymbol{\theta}) - \cos(\boldsymbol{\theta})\cos(t\boldsymbol{\theta})}{1-\cos^2(\boldsymbol{\theta})}$$

根据毕达哥拉斯三角恒等式（trigonometric Pythagorean identity）与

加法公式，我们有：

$$1-\cos^2(\boldsymbol{\theta})=\sin^2(\boldsymbol{\theta})$$
$$\cos((1-t)\boldsymbol{\theta})=\cos(\boldsymbol{\theta}-t\boldsymbol{\theta})=\cos(\boldsymbol{\theta})\cos(t\boldsymbol{\theta})+\sin(\boldsymbol{\theta})\sin(t\boldsymbol{\theta})$$
$$\sin((1-t)\boldsymbol{\theta})=\sin(\boldsymbol{\theta}-t\boldsymbol{\theta})=\sin(\boldsymbol{\theta})\cos(t\boldsymbol{\theta})-\cos(\boldsymbol{\theta})\sin(t\boldsymbol{\theta})$$

因此，

$$c_1=\frac{\cos(t\boldsymbol{\theta})-\cos(\boldsymbol{\theta})[\cos(\boldsymbol{\theta})\cos(t\boldsymbol{\theta})+\sin(\boldsymbol{\theta})\sin(t\boldsymbol{\theta})]}{\sin^2(\boldsymbol{\theta})}$$
$$=\frac{\cos(t\boldsymbol{\theta})-\cos(\boldsymbol{\theta})\cos(\boldsymbol{\theta})\cos(t\boldsymbol{\theta})-\cos(\boldsymbol{\theta})\sin(\boldsymbol{\theta})\sin(t\boldsymbol{\theta})}{\sin^2(\boldsymbol{\theta})}$$
$$=\frac{\cos(t\boldsymbol{\theta})(1-\cos^2(\boldsymbol{\theta}))-\cos(\boldsymbol{\theta})\sin(\boldsymbol{\theta})\sin(t\boldsymbol{\theta})}{\sin^2(\boldsymbol{\theta})}$$

$$=\frac{\cos(t\boldsymbol{\theta})\sin^2(\boldsymbol{\theta})-\cos(\boldsymbol{\theta})\sin(\boldsymbol{\theta})\sin(t\boldsymbol{\theta})}{\sin^2(\boldsymbol{\theta})}$$
$$=\frac{\sin(\boldsymbol{\theta})\cos(t\boldsymbol{\theta})-\cos(\boldsymbol{\theta})\sin(t\boldsymbol{\theta})}{\sin(\boldsymbol{\theta})}$$
$$=\frac{\sin((1-t)\boldsymbol{\theta})}{\sin(\boldsymbol{\theta})}$$

以及

$$c_2=\frac{\cos(\boldsymbol{\theta})\cos(t\boldsymbol{\theta})+\sin(\boldsymbol{\theta})\sin(t\boldsymbol{\theta})-\cos(\boldsymbol{\theta})\cos(t\boldsymbol{\theta})}{\sin^2(\boldsymbol{\theta})}$$
$$=\frac{\sin(t\boldsymbol{\theta})}{\sin(\boldsymbol{\theta})}$$

于是，我们就定义球面插值[4]（spherical interpolation）公式为：

$$\mathrm{slerp}(\boldsymbol{a},\boldsymbol{b},t)=\frac{\sin((1-t)\boldsymbol{\theta})\boldsymbol{a}+\sin(t\boldsymbol{\theta})\boldsymbol{b}}{\sin\boldsymbol{\theta}}$$，其中$t\in[0,1]$

将单位四元数看作4D单位向量，便可以解出四元数之间的夹角 $\boldsymbol{\theta}=\arccos(\boldsymbol{a}\cdot\boldsymbol{b})$。

如果$\boldsymbol{a}$与$\boldsymbol{b}$之间的夹角$\boldsymbol{\theta}$接近于0，那么$\sin\boldsymbol{\theta}$也接近于0，因此有限的数值精度会引起上式中除数为0的问题。在这种情况下，我们要在这两个四元数之间进行线性插值，再对结果执行规范化处理。此时，我们即可得到$\boldsymbol{\theta}$较小时的一个极为接近的插值结果（见图22.7）。

观察图22.8，其中展示的是待完成线性插值后，再将插值四元数投影回单位球面而导致的非线性旋转速率（nonlinear rate of rotation）。这是因为我们对夹角过大的四元数采用线性插值，从而导致其转速忽快忽慢（即线段ab上进行的是均匀线性插值，但投影至球面上却有很大差异）。这通常并不是我们所期待的效果，同时也从侧面反映出球面插值更受追捧的原因（球面插值使旋转速率保持恒定）。

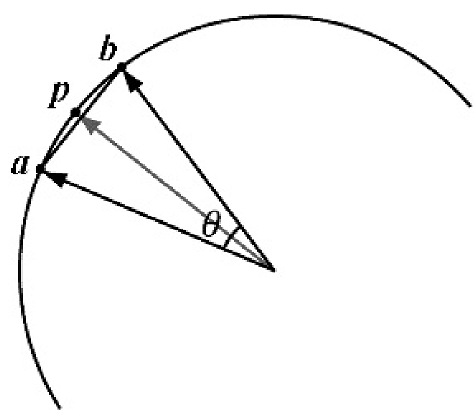


图22.7　对于$\boldsymbol{a}$与$\boldsymbol{b}$之间夹角θ较小的情况，通过四元数进行线性插值以及规范化线性插值即可得到与球面插值极为近似的结果。然而，当我们采用线性插值时，插值四元数却并不位于单位球面之上。所以我们必须对结果进行规范化处理，将它投影回单位球面上

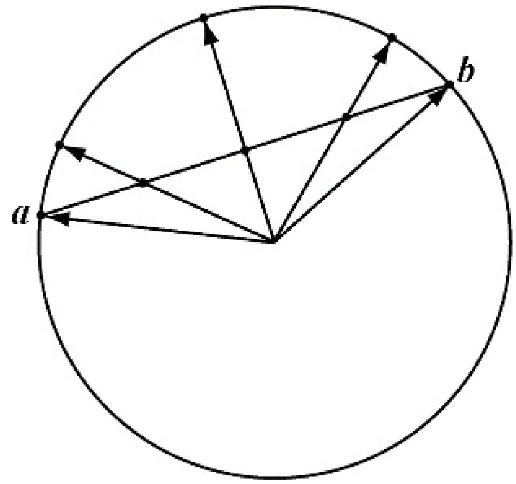


图22.8　在对四元数进行线性插值以及规范化处理后，我们却会得到单位球面上的非线性插值结果。这意味着旋转速率随着线性插值而加快或减慢，而并不是按恒速旋转

现在来讨论一种四元数的有趣性质。由于$(s\boldsymbol{q})* = s\boldsymbol{q}*$且标量与四元数乘法运算满足交换律，我们就有：

$$
\begin{aligned}
R_{-\mathrm{q}}(\boldsymbol{v}) &= -\boldsymbol{q}\boldsymbol{v}(-\boldsymbol{q})* \\
&= (-1)\boldsymbol{q}\boldsymbol{v}(-1)\boldsymbol{q}* \\
&= \boldsymbol{q}\boldsymbol{v}\boldsymbol{q}*
\end{aligned}
$$

因此，我们就得到了四元数$\boldsymbol{q}$与四元数$-\boldsymbol{q}$表示的是同一种旋转这一性质。我们再来以另一种方法对此进行验证，如果$\boldsymbol{q} = \left(\boldsymbol{n}\sin\frac{\boldsymbol{\theta}}{2}, \cos\frac{\boldsymbol{\theta}}{2}\right)$，那么

$$
\begin{aligned}
-\boldsymbol{q} &= \left(-\boldsymbol{n}\sin\frac{\boldsymbol{\theta}}{2}, -\cos\frac{\boldsymbol{\theta}}{2}\right) \\
&= \left(-\boldsymbol{n}\sin\left(\pi - \frac{\boldsymbol{\theta}}{2}\right), \cos\left(\pi - \frac{\boldsymbol{\theta}}{2}\right)\right) \\
&= \left(-\boldsymbol{n}\sin\left(\frac{2\pi - \boldsymbol{\theta}}{2}\right), \cos\left(\frac{2\pi - \boldsymbol{\theta}}{2}\right)\right)
\end{aligned}
$$

即，R_q绕轴$\boldsymbol{n}$旋转角$\boldsymbol{\theta}$，而R_{-q}绕轴$-\boldsymbol{n}$旋转角$2\pi - \boldsymbol{\theta}$。从几何学上来

看，位于4D单位球面上的单位四元数$\boldsymbol{q}$与其极坐标相反的单位四元数$-\boldsymbol{q}$表示着相同的方向。图22.9展示的就是这样的两个旋转结果相同的四元数。但是我们也可以轻易地看出两者的区别：一个旋转的角度较小，另一个旋转的角度过大。

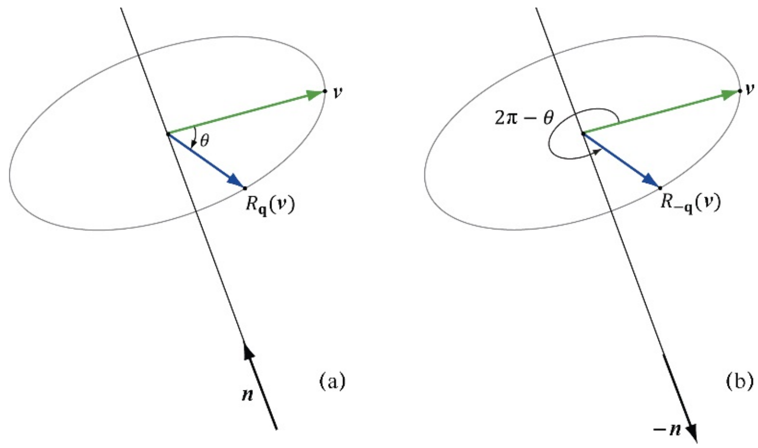


图22.9　R_q绕轴$\boldsymbol{n}$旋转角θ，R_{-q}绕轴$-\boldsymbol{n}$旋转角$2\pi-\theta$

由于四元数$\boldsymbol{b}$与$-\boldsymbol{b}$表示的是同样的方向，我们对四元数进行插值便有了$\mathrm{slerp}(\boldsymbol{a},\boldsymbol{b},t)$与$\mathrm{slerp}(\boldsymbol{a},-\boldsymbol{b},t)$两种选择。一种四元数的插值方法是以最小的旋转角度高效地完成旋转操作（类似于图22.9a所示的情形），另一种则需要绕很远的一段距离才能完成任务（与图22.9b中的情况相似）。现在我们再来看图22.10，我们希望根据略过较小弧度在4D单位球面上插值的这一标准，在四元数$\boldsymbol{b}$与$-\boldsymbol{b}$两者之间进行选择。如果选择了较小弧度的旋转方式，那么将采取最短、最直接的路径进行插值；如果选择了较大弧度的旋转方式，则会使物体多走出不少的冤枉路[Eberly01]，毕竟它旋转的路径太长了。

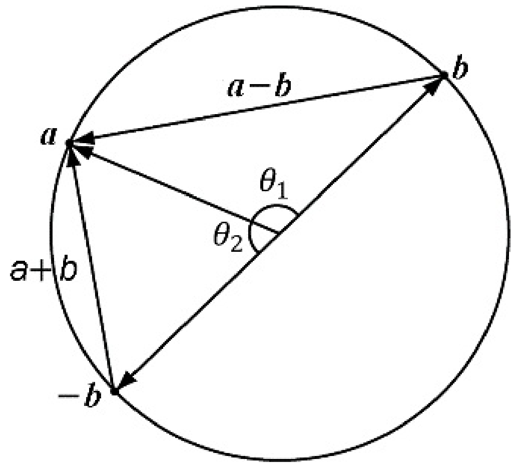


图22.10　从$\boldsymbol{a}$到$\boldsymbol{b}$进行插值即为按较大弧度θ_1在4D单位球面上插值，反之，从$\boldsymbol{a}$到$-\boldsymbol{b}$进行插值则为按较小弧度θ_2在4D单位球面上插值。当然，我们总是希望按最小的弧度，以最短、最直接的路径在4D单位球面上插值

根据[Watt92]，要找到表示按最小弧度在4D单位球面上旋转的四元数，我们就要对$\|\boldsymbol{a}-\boldsymbol{b}\|^2$与$\|\boldsymbol{a}-(-\boldsymbol{b})\|^2=\|\boldsymbol{a}+\boldsymbol{b}\|^2$进行比较。如果$\|\boldsymbol{a}+\boldsymbol{b}\|^2<\|\boldsymbol{a}-\boldsymbol{b}\|^2$，那么就选择以$-\boldsymbol{b}$来代替$\boldsymbol{b}$进行插值，因为这时$-\boldsymbol{b}$更接近于$\boldsymbol{a}$，这样会使旋转的弧度更小，经过的路径（弧长）更短。

```
// 线性插值（针对theta较小的情况）
public static Quaternion LerpAndNormalize(Quaternion p, Quaternion q, floa
t s)
{
  // 对结果进行规范化处理以使其成为单位四元数
  return Normalize((1.0f - s)*p + s*q);
}

public static Quaternion Slerp(Quaternion p, Quaternion q, float s)
{
  // 前面曾讲到四元数q与-q表示着相同的方向，但是这两者的插值过程却不相同：一种要
略过最小的弧，
  // 另一种将绕过较大的弧长。而为了找到最小的弧长，就要对p-q的模与p-(-q) = p+q
的模进行比较

  if(LengthSq(p-q) > LengthSq(p+q))
    q = -q;

  float cosPhi = DotP(p, q);
```

```
  // 对于角度极小的情况，采用线性插值
  if(cosPhi > (1.0f - 0.001))
    return LerpAndNormalize(p, q, s);

  // 求出两个四元数之间的夹角
  float phi = (float)Math.Acos(cosPhi);

  float sinPhi = (float)Math.Sin(phi);

  // 沿着p，q与单位球原点所在平面与4D单位球面的交点所围成的弧进行插值
  return ((float)Math.Sin(phi*(1.0-s))/sinPhi)*p +
    ((float)Math.Sin(phi*s)/sinPhi)*q;
}
```

22.5 DirectX数学库中与四元数有关的函数

DirectX数学库支持四元数的相关运算。由于四元数的“数据”是4个实数，所以DirectX数学库以XMVECTOR类型来存储四元数。以下是该库中一些常用的四元数函数定义。

```
// 返回四元数点积Q1·Q2
XMVECTOR XMQuaternionDot(XMVECTOR Q1, XMVECTOR Q2);

// 返回四元数单位元(0, 0, 0, 1)
XMVECTOR XMQuaternionIdentity();

// 返回四元数Q的共轭四元数
XMVECTOR XMQuaternionCojugate(XMVECTOR Q);

// 返回四元数Q的范数（模）
XMVECTOR XMQuaternionLength(XMVECTOR Q);

// 把四元数Q作为一个4D向量进行规范化处理
XMVECTOR XMQuaternionNormalize(XMVECTOR Q);

// 计算四元数的乘积Q1Q2
XMVECTOR XMQuaternionMultiply(XMVECTOR Q1, XMVECTOR Q2);

// 根据旋转轴及旋转角表示法来计算相应的四元数
XMVECTOR XMQuaternionRotationAxis(XMVECTOR Axis, FLOAT Angle);

// 根据旋转轴与旋转角表示法来计算对应的四元数，这里的旋转轴是一个规范化向量——因此该方法的执行
// 速应快于XMQuaternionRotationAxis方法
XMVECTOR XMQuaternionRotationNormal(XMVECTOR NormalAxis,FLOAT Angle);

// 根据给定的旋转矩阵来计算相应的四元数
XMVECTOR XMQuaternionRotationMatrix(XMMATRIX M);

// 根据给定的单位四元数来构建对应的旋转矩阵[5]
XMMATRIX XMMatrixRotationQuaternion(XMVECTOR Quaternion);

// 从四元数Q中提取旋转轴以及关于此轴的旋转角
VOID XMQuaternionToAxisAngle(XMVECTOR *pAxis, FLOAT *pAngle, XMVECTOR Q);
```

```
// 返回slerp(Q1, Q2, ![{t}](http://private.codecogs.com/gif.latex?{t}))的计算结果，输入的两个四元数必为单位四元数
XMVECTOR XMQuaternionSlerp(XMVECTOR Q0, XMVECTOR Q1, FLOAT t);
```

22.6　旋转演示程序

在本章的演示程序中，我们要使一个骷髅头网格围绕简易的场景移动。在整个过程中，此网格的位置、朝向以及大小都是时刻变化着的。我们通过四元数来表示骷髅头的朝向，运用四元数球面插值函数slerp在方向之间进行插值，并以线性插值对网格的位置与大小插值。此例程也为第23章的主题“角色动画”起到“预热”的效果。

关键帧动画（key frame animation）是一种常见动画的形式。一个关键帧（key frame）指定了物体在某一时刻的位置、朝向以及大小。本章示例（位于AnimationHelper.h/.cpp文件内）中定义了下列关键帧结构体。

```
struct Keyframe
{
  Keyframe();
  ~Keyframe();

  float TimePos;
  XMFLOAT3 Translation;
  XMFLOAT3 Scale;
  XMFLOAT4 RotationQuat;
};
```

动画（animation）即为按时间排序的一系列关键帧。

```
struct BoneAnimation
{
  float GetStartTime()const;
  float GetEndTime()const;

  void Interpolate(float t, XMFLOAT4X4& M)const;
```

```
  std::vector<Keyframe> Keyframes;
};
```

“bone”这个名字的由来将在下一节中揭晓。现在我们只需知道驱动一个单独的“bone”就如同驱动一个单独的物体。GetStartTime方法仅返回第一个关键帧的起始时间点。比如说，一个物体可能在时间轴的前10秒内都不会动，这时，此函数便派上了用场。类似地，GetEndTime方法返回的则是最后一个关键帧的结束时间点，这便于我们了解动画的结尾来停止驱动“bone”。

我们现在已经有了一系列的关键帧，它们定义了一部看起来有些粗糙的动画。但是，关键帧之间的帧又该怎样来实现呢？这又该轮到插值计算大显神威了。针对两个关键帧K_i与K_{i+1}之间的不同时刻t，我们在这两个关键帧之间对此进行插值。

```
void BoneAnimation::Interpolate(float t, XMFLOAT4X4& M)const
{
  // 由于t是动画开始前的时刻，所以仅返回第一个关键帧
  if( t <= Keyframes.front().TimePos )
  {
   XMVECTOR S = XMLoadFloat3(&Keyframes.front().Scale);
   XMVECTOR P = XMLoadFloat3(&Keyframes.front().Translation);
   XMVECTOR Q = XMLoadFloat4(&Keyframes.front().RotationQuat);

   XMVECTOR zero = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
   XMStoreFloat4x4(&M, XMMatrixAffineTransformation(S, zero, Q, P));
  }
  // 由于t是动画结束后的时刻，因此仅返回最后一个关键帧
  else if( t >= Keyframes.back().TimePos )
  {
   XMVECTOR S = XMLoadFloat3(&Keyframes.back().Scale);
   XMVECTOR P = XMLoadFloat3(&Keyframes.back().Translation);
   XMVECTOR Q = XMLoadFloat4(&Keyframes.back().RotationQuat);

   XMVECTOR zero = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
   XMStoreFloat4x4(&M, XMMatrixAffineTransformation(S, zero, Q, P));
```

```
    }
    // t位于两个关键帧之间，所以对此进行插值
    else
    {
     for(UINT i = 0; i < Keyframes.size()-1; ++i)
     {
       if( t >= Keyframes[i].TimePos && t <= Keyframes[i+1].TimePos )
       {
        float lerpPercent = (t - Keyframes[i].TimePos) /
          (Keyframes[i+1].TimePos - Keyframes[i].TimePos);

        XMVECTOR s0 = XMLoadFloat3(&Keyframes[i].Scale);
        XMVECTOR s1 = XMLoadFloat3(&Keyframes[i+1].Scale);

        XMVECTOR p0 = XMLoadFloat3(&Keyframes[i].Translation);
        XMVECTOR p1 = XMLoadFloat3(&Keyframes[i+1].Translation);

        XMVECTOR q0 = XMLoadFloat4(&Keyframes[i].RotationQuat);
        XMVECTOR q1 = XMLoadFloat4(&Keyframes[i+1].RotationQuat);

        XMVECTOR S = XMVectorLerp(s0, s1, lerpPercent);
        XMVECTOR P = XMVectorLerp(p0, p1, lerpPercent);
        XMVECTOR Q = XMQuaternionSlerp(q0, q1, lerpPercent);

        XMVECTOR zero = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
        XMStoreFloat4x4(&M, XMMatrixAffineTransformation(S, zero, Q, P));

        break;
       }
     }
    }
}
```

图22.11展示了在关键帧**1**与关键帧**2**之间进行插值所生成的中间帧（in-between frame）。

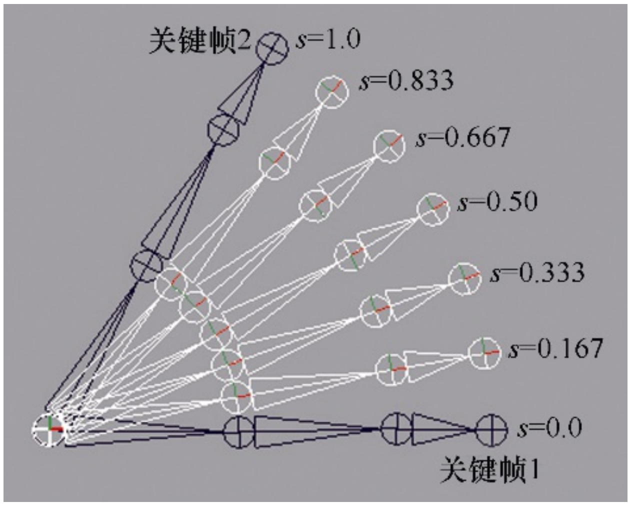


图22.11　关键帧插值计算的示意图。关键帧定义了动画中的“关键”姿态，
而关键帧之间的插值则表示其间的中间帧

注意　Note

图22.11曾用于Frank D. Luna的图书，Introduction to 3D Game Programming with DirectX 9.0c: A Shader Approach, 2006: Jones and Bartlett Learning, Burlington, MA. www.jblearning.com，现经许可摘录对此。

插值完成之后，我们还要在着色器程序中利用矩阵对其进行变换，所以还需构建对应的变换矩阵。这里所用的XMMatrixAffineTransformation函数声明如下。

```
XMMATRIX XMMatrixAffineTransformation(
  XMVECTOR Scaling,
  XMVECTOR RotationOrigin,
```

```
  XMVECTOR RotationQuaternion,
  XMVECTOR Translation);
```

至此，简易动画系统已经准备就绪。接下来的工作就是定义一些关键帧。

```
// App类中的成员数据

float mAnimTimePos = 0.0f;
BoneAnimation mSkullAnimation;
//
// In constructor,define the animation keyframes
//
void QuatApp::DefineSkullAnimation()
{
  //
  // 定义动画的关键帧
  //

  XMVECTOR q0 = XMQuaternionRotationAxis(
    XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f), XMConvertToRadians(30.0f));
  XMVECTOR q1 = XMQuaternionRotationAxis(
    XMVectorSet(1.0f, 1.0f, 2.0f, 0.0f), XMConvertToRadians(45.0f));
  XMVECTOR q2 = XMQuaternionRotationAxis(
    XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f), XMConvertToRadians(-30.0f));
  XMVECTOR q3 = XMQuaternionRotationAxis(
    XMVectorSet(1.0f, 0.0f, 0.0f, 0.0f), XMConvertToRadians(70.0f));

  mSkullAnimation.Keyframes.resize(5);
  mSkullAnimation.Keyframes[0].TimePos = 0.0f;
  mSkullAnimation.Keyframes[0].Translation = XMFLOAT3(-7.0f, 0.0f,
    0.0f);
  mSkullAnimation.Keyframes[0].Scale = XMFLOAT3(0.25f, 0.25f, 0.25f);
  XMStoreFloat4(&mSkullAnimation.Keyframes[0].RotationQuat, q0);

  mSkullAnimation.Keyframes[1].TimePos = 2.0f;
  mSkullAnimation.Keyframes[1].Translation = XMFLOAT3(0.0f, 2.0f,
    10.0f);
  mSkullAnimation.Keyframes[1].Scale = XMFLOAT3(0.5f, 0.5f, 0.5f);
  XMStoreFloat4(&mSkullAnimation.Keyframes[1].RotationQuat, q1);

  mSkullAnimation.Keyframes[2].TimePos = 4.0f;
  mSkullAnimation.Keyframes[2].Translation = XMFLOAT3(7.0f, 0.0f,
    0.0f);
  mSkullAnimation.Keyframes[2].Scale = XMFLOAT3(0.25f, 0.25f, 0.25f);
```

```
  XMStoreFloat4(&mSkullAnimation.Keyframes[2].RotationQuat, q2);

  mSkullAnimation.Keyframes[3].TimePos = 6.0f;
  mSkullAnimation.Keyframes[3].Translation = XMFLOAT3(0.0f, 1.0f,
    -10.0f);
  mSkullAnimation.Keyframes[3].Scale = XMFLOAT3(0.5f, 0.5f, 0.5f);
  XMStoreFloat4(&mSkullAnimation.Keyframes[3].RotationQuat, q3);

  mSkullAnimation.Keyframes[4].TimePos = 8.0f;
  mSkullAnimation.Keyframes[4].Translation = XMFLOAT3(-7.0f, 0.0f,
    0.0f);
  mSkullAnimation.Keyframes[4].Scale = XMFLOAT3(0.25f, 0.25f, 0.25f);
  XMStoreFloat4(&mSkullAnimation.Keyframes[4].RotationQuat, q0);
}
```

每一个关键帧中都将改变骷髅头的大小，并将它以不同朝向安置在场景中的不同位置。如果读者有兴趣，可以尝试为此演示程序添加自己所构思的关键帧或修改关键帧数据。例如，读者可以把所有帧的旋转与缩放变换都设为恒等变换（identity，也就是变换后却保持原有状态），以此来观看只有物体位置发生变化的动画效果。

最后一个步骤是随着时间的改变，通过对当前的骷髅头世界矩阵进行插值来确定此帧的动画数据。

```
void QuatApp::UpdateScene(float dt)
{
  ...

  // 使时间的车轮向前滚动（推进时间点在时间轴上的位置）
  mAnimTimePos += dt;
  if(mAnimTimePos >= mSkullAnimation.GetEndTime())
  {
    // 将动画环回至起始时间
    mAnimTimePos = 0.0f;
  }

  // 求出此刻的骷髅头世界矩阵
  mSkullAnimation.Interpolate(mAnimTimePos, mSkullWorld);

  ...
```

```
}
```

为了使骷髅头“摇头晃脑”，令它的世界矩阵在每一帧都有所变化，如图22.12所示。

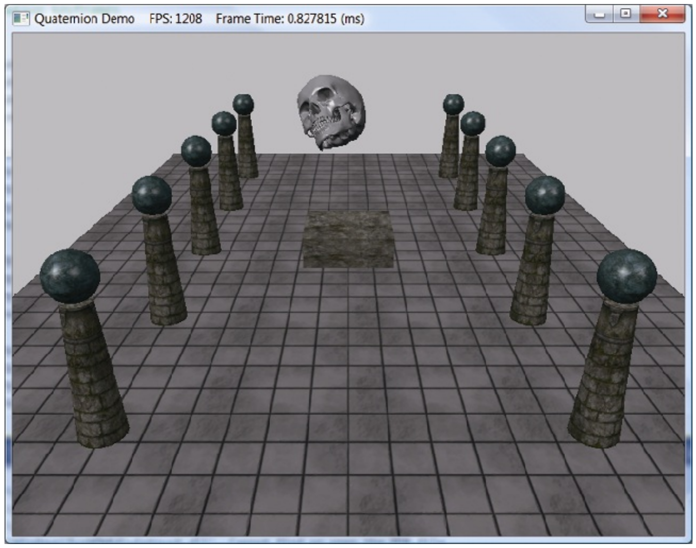


图22.12　四元数演示程序的效果

22.7　小结

1．一个有序实数四元组$\boldsymbol{q}=(x,y,z,w)=(q_1,q_2,q_3,q_4)$即为一个四元数，它通常被简记为$\boldsymbol{q}=(\boldsymbol{u},w)=(x,y,z,w)$，我们称$\boldsymbol{u}=(x,y,z)$为向量虚部，$w$为实部。而且，我们将四元数的相等以及加、减、乘三则运算分别定义如下：

（a）$(\boldsymbol{u},\ a)=(\boldsymbol{v},\ b)$，当且仅当$\boldsymbol{u}=v$且$a=b$。

（b）$(\boldsymbol{u},\ a)\pm(\boldsymbol{v},\ b)=(\boldsymbol{u}\pm\boldsymbol{v},\ a\pm b)$。

（c）$(\boldsymbol{u},\ a)(\boldsymbol{v},\ b)=(a\boldsymbol{v}+b\boldsymbol{u}+\boldsymbol{u}\times\boldsymbol{v},\ ab-\boldsymbol{u}\cdot\boldsymbol{v})$。

2．四元数之间的乘法运算不满足交换律，但是满足结合律。可将四元数$\boldsymbol{e}=(0,0,0,1)$看作乘法单位元。四元数乘法对加法的分配律为$\boldsymbol{p}(\boldsymbol{q}+\boldsymbol{r})=\boldsymbol{pq}+\boldsymbol{pr}$以及$(\boldsymbol{q}+\boldsymbol{r})\boldsymbol{p}=\boldsymbol{qp}+\boldsymbol{rp}$。

3．我们可以通过把实数s写作$s=(0,0,0,s)$来将其转换到四元数空间，并以$\boldsymbol{u}=(\boldsymbol{u},0)$的表示方法将向量$\boldsymbol{u}$转换至四元数空间。一个实部为0的四元数称为纯四元数（pure quaternion）。标量与四元数的乘法为$s(p_1,p_2,p_3,p_4)=(sp_1,sp_2,sp_3,sp_4)=(p_1,p_2,p_3,p_4)s$。因此，四元数与标量乘法是四元数乘法运算中满足交换律的一种特殊情况。

4．四元数$\boldsymbol{q}=(q_1,q_2,q_3,q_4)=(\boldsymbol{u},q_4)$的共轭四元数记作$\boldsymbol{q}*$，它的定义为$\boldsymbol{q}*=q_1-q_2-q_3+q_4=(-\boldsymbol{u},q_4)$。四元数的范数（norm。或称模，即

magnitude）定义为$\|\boldsymbol{q}\| = \sqrt{\boldsymbol{q}\boldsymbol{q}*} = \sqrt{q_1^2 + q_2^2 + q_3^2 + q_4^2} = \sqrt{\|\boldsymbol{u}\|^2 + q_4^2}$。如果一个四元数的范数为1，那么称之为单位四元数（unit quaternion）。

5．设$\boldsymbol{q} = (q_1, q_2, q_3, q_4) = (\boldsymbol{u}, q_4)$为一个非零四元数，那么，它的逆记作$\boldsymbol{q}^{-1}$，且定义为$\boldsymbol{q}^{-1} = \dfrac{\boldsymbol{q}*}{\boldsymbol{q}^2}$。如果$\boldsymbol{q}$是一个单位四元数，那么$\boldsymbol{q}^{-1} = \boldsymbol{q}*$。

6．可以把一个单位四元数$\boldsymbol{q} = (\boldsymbol{u}, q_4)$写作极坐标的表示形式（polar representation）$\boldsymbol{q} = (\sin \boldsymbol{\theta} \boldsymbol{n}, \cos \boldsymbol{\theta})$，其中的$\boldsymbol{n}$为一个单位向量。

7．如果$\boldsymbol{q}$为一个单位四元数，那么当$\|\boldsymbol{n}\| = 1$且$\boldsymbol{\theta} \in [0, \pi]$时，则有$\boldsymbol{q} = (\sin \boldsymbol{\theta} \boldsymbol{n}, \cos \boldsymbol{\theta})$。四元数旋转算子的定义为$R_{\mathrm{q}}(\boldsymbol{v}) = \boldsymbol{q}\boldsymbol{v}\boldsymbol{q}^{-1} = \boldsymbol{q}\boldsymbol{v}\boldsymbol{q}*$，它表示点或向量$\boldsymbol{v}$绕轴$\boldsymbol{n}$按角$2\boldsymbol{\theta}$进行旋转。$R_q$有相应的矩阵表示法，而且任何的旋转矩阵都能被转换为一个表示其对应旋转操作的四元数。

8．对于动画方面而言，在两个方向之间进行插值是一项很常见的工作。把每个方向都用一个单位四元数表示，我们就能通过对这些单位四元数进行球面插值来求取相应的插值方向。

22.8 练习

1．计算下列复数算式。

（a）$(3+2i)+(-1+i)$

（b）$(3+2i)-(-1+i)$

（c）$(3+2i)(-1+i)$

（d）$4(-1+i)$

（e）$(3+2i)/(-1+i)$

（f）$\boldsymbol{z}=(3+2i)$，计算$\boldsymbol{z}$的共轭复数$\bar{\boldsymbol{z}}$。

（g）$|3+2i|$

2．把复数$(-1,\ 3)$写作极坐标的形式。

3．通过复数的乘法运算使向量(2, 1)旋转$30°$。

4．根据复数除法的定义来证明：$\dfrac{a+ib}{a+ib}=1$。

5．设$\boldsymbol{z}=a+ib$。证明$|\boldsymbol{z}|^2=\boldsymbol{z}\bar{\boldsymbol{z}}$。

6．设$\boldsymbol{M}$为一个2×2矩阵。证明：$\det\boldsymbol{M}=1$且$\boldsymbol{M}^{-1}=\boldsymbol{M}^{\mathrm{T}}$，当且仅

当$M = \begin{bmatrix} \cos\boldsymbol{\theta} & \sin\boldsymbol{\theta} \\ -\sin\boldsymbol{\theta} & \cos\boldsymbol{\theta} \end{bmatrix}$。即，当且仅当$M$是一个旋转矩阵。这给验证某矩阵是否为旋转矩阵提供了一条途径。

7．设$\boldsymbol{p} = (1, 2, 3, 4)$且$\boldsymbol{q} = (2, -1, 1, -2)$都为四元数。计算下列各式。

（a）$\boldsymbol{p} + \boldsymbol{q}$

（b）$\boldsymbol{p} - \boldsymbol{q}$

（c）$\boldsymbol{pq}$

（d）$\boldsymbol{p}*$

（e）$\boldsymbol{q}*$

（f）$\boldsymbol{p} * \boldsymbol{p}$

（g）$||\boldsymbol{p}||$

（h）$||\boldsymbol{q}||$

（i）$\boldsymbol{p}^{-1}$

（j）$\boldsymbol{q}^{-1}$

8．将单位四元数$\boldsymbol{q} = \left(\frac{1}{2}, \frac{1}{2}, 0, \frac{1}{\sqrt{2}}\right)$写作极坐标的形式。

9．把单位四元数$\boldsymbol{q}=\left(\frac{\sqrt{3}}{2},0,0,-\frac{1}{2}\right)$写作极坐标的形式。

10．求出令向量（或点）绕轴$(1,1,1)$旋转$45°$的单位四元数。

11．求出令向量（或点）绕轴$(0,\ 0,\ -1)$旋转$60°$的单位四元数。

12．设$\boldsymbol{p}=\left(\frac{1}{2},0,0,\frac{\sqrt{3}}{2}\right)$与$\boldsymbol{q}=\left(\frac{\sqrt{3}}{2},0,0,\frac{1}{2}\right)$皆为单位四元数。计算$\mathrm{slerp}\left(\boldsymbol{p},\boldsymbol{q},\frac{1}{2}\right)$并验证结果也为一个单位四元数。

13．证明可以将四元数(x,y,z,w)写作$x\boldsymbol{i}+y\boldsymbol{j}+z\boldsymbol{k}+w$的形式。

14．证明$qq*=q*q=q_1^2+q_2^2+q_3^2+q_4^2=\|u\|^2+q_4^2$。

15．设$\boldsymbol{p}=(\boldsymbol{u},0)$与$\boldsymbol{q}=(\boldsymbol{v},0)$为纯四元数（即实部为0）。证明$\boldsymbol{pq}=(\boldsymbol{p}\times\boldsymbol{q},-\boldsymbol{p}\cdot\boldsymbol{q})$。

16．证明下列性质。

（a）$(\boldsymbol{pq})*=\boldsymbol{q}*\boldsymbol{p}*$

（b）$(\boldsymbol{p}+\boldsymbol{q})*=\boldsymbol{p}*+\boldsymbol{q}*$

（c）当$s\in\mathbb{R}$时，$(s\boldsymbol{q}*)*=s\boldsymbol{q}*$

（d）$\boldsymbol{qq}*=\boldsymbol{q}*\boldsymbol{q}=\boldsymbol{q}_1^2+\boldsymbol{q}_2^2+\boldsymbol{q}_3^2+\boldsymbol{q}_4^2$

（e）$\|\boldsymbol{pq}\|=\|\boldsymbol{p}\|\,\|\boldsymbol{q}\|$

17．用代数方法证明：$\boldsymbol{a}\cdot\dfrac{\sin((1-t)\boldsymbol{\theta})\boldsymbol{a}+\sin(t\boldsymbol{\theta})\boldsymbol{b}}{\sin\boldsymbol{\theta}}=\cos(t\boldsymbol{\theta})$

18．设$\boldsymbol{a}$、$\boldsymbol{b}$、$\boldsymbol{c}$都为3D向量。证明下述恒等式：

（a）$\boldsymbol{a}\times(\boldsymbol{b}\times\boldsymbol{c})=(\boldsymbol{a}\cdot\boldsymbol{c})\boldsymbol{b}-(\boldsymbol{a}\cdot\boldsymbol{b})\boldsymbol{c}$

（b）$(\boldsymbol{a}\times\boldsymbol{b})\times\boldsymbol{c}=-(\boldsymbol{c}\cdot\boldsymbol{b})\boldsymbol{a}+(\boldsymbol{c}\cdot\boldsymbol{a})\boldsymbol{b}$

---

[1]　本章中，作者在四元数的表示方法上似乎有点不按套路出牌，即与常见的文献稍有差异，但原理和计算方法是一致的。看起来，作者希望用矩阵、向量、三角函数与类比法来讲解，这样会绕过数学上烦琐的形式化证明，更易理解。另外，有些地方的跨度可能比较大，读者可参考一些其他的资料来加以补充（基本上回看开篇的向量与矩阵知识即可），这也就是作者要在开篇向读者推荐专业书籍的原因吧，本书里的四元数充其量只是一种工具而已。上文推荐的那本书的作者有篇论文名为《quaternions and rotation sequences》（即“四元数及其旋转序列”，与原书同名），可用作原书的基本概述并作为补充材料。另有维基百科可参考查阅。

[2]　由于虚部是向量，所以此处直接看作向量叉积运算即可，下同。

[3]　推导过程中还运用了叉积的反交换律$a\times b=-b\times a$与点积的交换律$a\cdot b=b\cdot a$。

[4]　事实上，这种插值方式通常被称为球面线性插值，即spherical linear interpolation，缩写为slerp。

[5] 官方注释是“根据一个四元数来构建旋转矩阵”。

第23章　角色动画

注意 Note

本章的部分内容曾出现在Frank D. Luna的著作Introduction to 3D Game Programming with DirectX 9.0c: A Shader Approach, 2006: Jones and Bartlett Learning, Burlington, MA，现经许可摘录。

在本章中，我们将学习如何驱使人类或动物这些复杂的动画角色运动起来。所谓复杂，是因为需要在同一时刻令其中的许多可运动部分活动起来。回忆一下我们跑步的动作就会发现——每块骨骼（bone，是不是有点眼熟？）都要按各自的轨迹运动。不借助任何工具赤手空拳地创建这样复杂的动画是不切实际的，因此，要完成这项任务往往需要运用特定的模型与动画制作工具。假设现在已经拥有了角色及其相应的动画数据，让我们马上开始本章中用Direct3D令角色活灵活现以及用渲染动画的旅程吧。

学习目标：

1．熟悉与动态蒙皮网格相关的专业术语。

2．学习网格层次变换的数学描述，并对基于树型的网格层次结构进行遍历。

3．理解顶点混合技术的主体思想及其数学表示。

4．了解如何从文件中加载动画数据。

5．探索如何用Direct3D来实现角色动画。

23.1　框架层次[1]

许多物体是由保持“父子”关系的多个部分组成的，其中，存在一个或一个以上的物体子对象[2]能够按照它们自己的轨迹独立地运动（可能还要受到物理运动上的限制——比如，人类的关节部分只能在特定的范围内运动）。但是，当父对象运动时，其子对象也将被迫随之移动。举个例子，思考一下，上肢可自然地分为上臂、前臂与手这3个部分。手可以绕着腕关节独立于臂转动。但是，如果前臂绕着肘关节转动，则手也必随之转动。类似地，如果上臂绕着肩关节转动，那么前臂必随之转动，又由于前臂被带动，因而手亦随之运动（见图23.1）。这样一来，我们就得到了一种确切的对象层次结构（object hierarchy）关系：手是前臂的子对象，前臂又是上臂的子对象。如果再将此实例进行扩展，那么上臂将作为躯干的子对象……若以此类推，那么将得到一副完整的骨骼系统（图23.2展示了一种更为复杂的层次示例）。

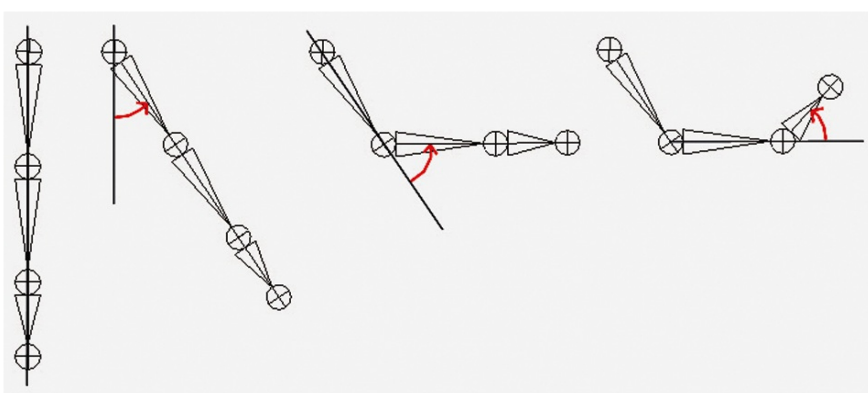


图23.1　层次变换（hierarchy transform）。可以观察到一块骨骼的父对象的变换会对它自己及其子对象造成影响

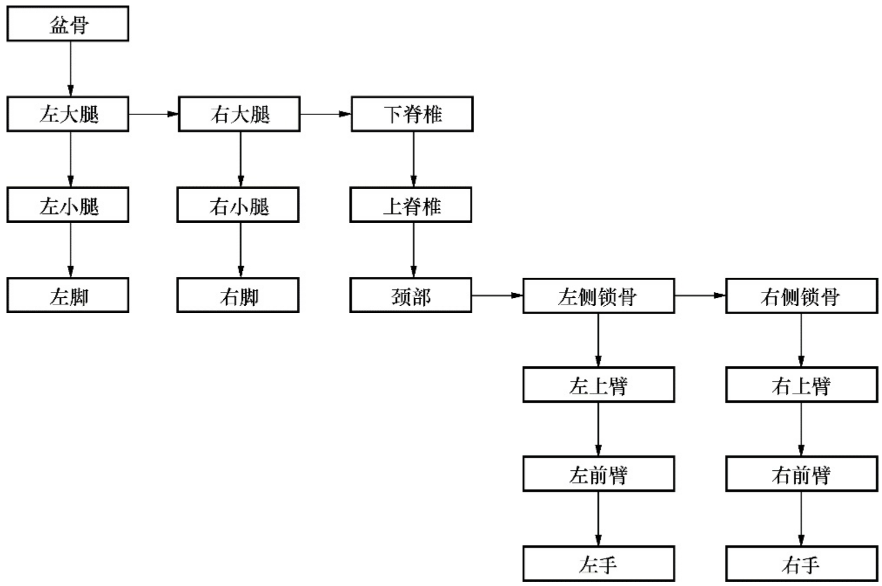


图23.2　一种模拟具有无毛两足动物基本根性角色的更为复杂的树状层次结构。“父对象”以向下的箭头指向它的“第一个子对象”，右向箭头则表示两对象之间的“兄弟”关系。例如，“左大腿”“右大腿”与“下脊椎”都为“盆骨”骨骼的子对象

本节的目标是展示如何根据一个对象及其祖先（ancestor，即它的父对象、祖父对象以及曾祖父对象等）对象的位置，将其正确地定位于场景之中。

数学描述

注意 Note

在学习本小节之前，读者不妨先回顾本书的第3章，特别是其中的坐标变换部分。

为了保证讲解过程简单而又具体，我们采用上臂（根对象）、前臂与手这一层次关系作为示例。在此，我们分别把它们标记为骨骼0、骨骼1以及骨骼2（见图23.3）。

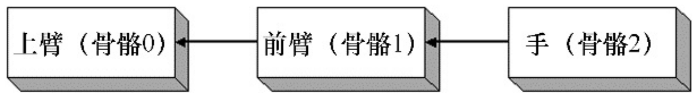


图23.3　一个简单的上肢层次结构图

只要理解了基本的概念，我们就可以把它直接推广到处理更为复杂的情景之中。当前的问题是，如果给定了某层次结构中的特定对象，如何把它正确地变换到世界空间中呢？显而易见的是，我们无法将该对象直接变换至世界空间，这是因为它在场景中的位置也受到其祖先对象的影响，所以我们还必须顾及那些祖先对象的变换。

层次结构中的每个对象都相对于其旋转所依赖的枢纽关节（pivot joint）为原点的局部坐标系进行建模（见图23.4）。

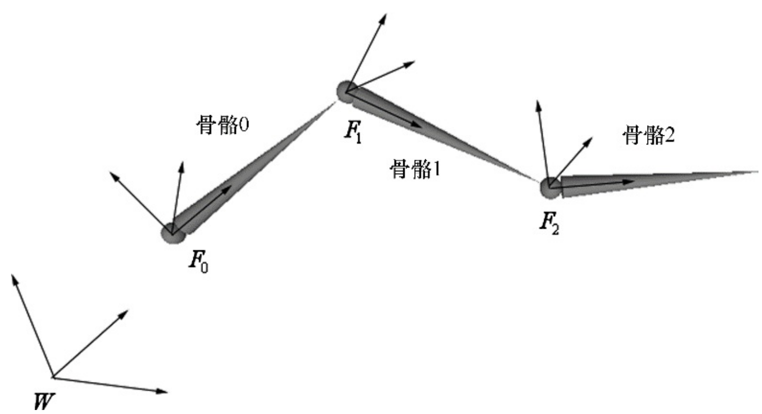


图23.4　每块骨骼几何体都是相对于其自身的局部坐标系来描述的。但是，由于所有的坐标系都位于同一个世界空间之中，所以我们就能够得到它们之间任意一对的联系

由于所有对象的坐标系都位于同一世界空间之中，所以我们就能基

于这一点将它们联系起来，特别是我们能够描述出：在任意时刻，每个对象坐标系与其父对象坐标系[3]的位置关系（这里研究的是固定时刻的物体快照，这是因为在一般情况下这些网格的各个层次都是相对活动的，而且它们的关系会随着一个特定的时间函数而发生改变）。（根对象标架，root frame，也有译作根框架，继而有框架F_1、框架F_2等）F_0的父坐标系为世界空间坐标系W，即坐标系F_0是相对于世界空间坐标来进行描述的。）掌握了父子坐标系之间的关系后，我们就可以用变换矩阵将坐标由子（对象）空间变换至其父（对象）空间（此思路与局部空间至世界空间的变换相一致，区别仅为这里执行的是局部空间向其父空间的变换）。设$\boldsymbol{A}_2$为将几何体从标架F_2变换至F_1的矩阵，$\boldsymbol{A}_1$为将几何体从标架F_1变换至F_0的矩阵，$\boldsymbol{A}_0$为将几何体从标架F_0变换至W的矩阵。（我们称$\boldsymbol{A}_i$为至父坐标系矩阵（to-parent），利用它可以将几何体从子坐标系变换至其父坐标系。）这样一来，我们就能通过下面定义的矩阵$\boldsymbol{M}_i$将上肢层次结构中的第i个对象变换至世界空间：

$$\boldsymbol{M}_i = \boldsymbol{A}_i \boldsymbol{A}_{i-1} \ldots \boldsymbol{A}_1 \boldsymbol{A}_0 \qquad (23.1)$$

尤其是在上肢实例中，利用矩阵$\boldsymbol{M}_2 = \boldsymbol{A}_2 \boldsymbol{A}_1 \boldsymbol{A}_0$，$\boldsymbol{M}_1 = \boldsymbol{A}_1 \boldsymbol{A}_0$与$\boldsymbol{M}_0 = \boldsymbol{A}_0$就能分别将手部、前臂与上臂直接变换到世界空间。可以观察到，每一个对象都继承了其祖先对象的变换操作，这也就满足了上文所提到的如果上臂移动，那么手部也随之运动这一要求。

图23.5用几何的方式详细解释了式（23.1）。事实上，要对上肢层次结构中的某个对象进行变换，只需依次应用该对象及其各祖先的“至父坐标系矩阵”，使之按其祖先对象升序的次序在坐标系层次中上滤

（percolate up），直到此对象变换至世界空间为止。

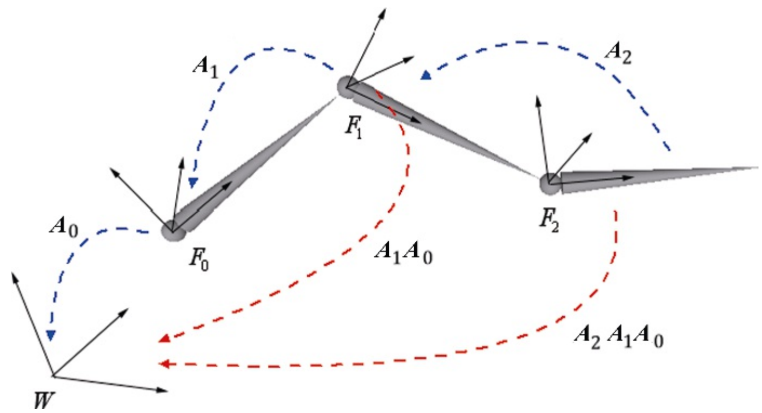


图23.5　由于可以在同一个世界空间中描述层次结构里各对象坐标系的关系，所以我们能够将某个
对象变换至另一个对象的空间中。特别是在上肢层次结构这个示例中，我们就是以骨骼坐标系及其
父坐标系的关系建立起坐标系之间的联系。据此就能构建出至父坐标系的变换矩阵，以此将骨骼
几何体从它的局部空间变换至其父坐标系。一旦该对象位于其父坐标系，我们就能通过其父
对象的“至父坐标系矩阵”，再将此对象变换至其祖父坐标系之中，并以此类推，
直到此对象遍历每个祖先对象的坐标系，变换至世界空间之中

在上肢层次结构这个示例中，我们采用了简单的线性层次结构来描述其中对象的关系。这种想法也可以推广到一种树形的层次结构。在这种结构中，将对象变换至世界空间的方式变为了依次应用该对象及其各祖先（按升序顺序）的“至父坐标系矩阵”使之在层次结构中上滤，直至目标对象变换至世界空间为止（这里再强调一次，根标架的父坐标系即为世界空间）。这唯一的实质性差别即我们用更为复杂的树形结构取代了此前的线性链表结构。

我们再来看一个更复杂的示例，考虑图23.2所示的角色左侧锁骨。

由于它是颈骨的同级兄弟对象，因此它也是上脊椎的一个子对象。这样一来，左侧锁骨的世界变换是经过左侧锁骨至父坐标系的变换、上脊椎至父坐标系的变换、下脊椎至父坐标系的变换、盆骨至父坐标系的变换来实现的。

23.2　蒙皮网格

23.2.1　定义

图23.6所示的是一款动画角色网格。图中高亮标出的骨骼链称为骨架（skeleton），而它所具有的性质使它为驱动一款角色动画系统的工作提供了一份天然的层次结构。骨架被外表皮（skin）附着，我们用3D几何图形（顶点与多边形）来对它进行模拟，这便是蒙皮操作。皮肤顶点所位于的绑定空间（bind space）是定义整个皮肤时所参照的局部坐标系（通常是根坐标系）。与现实中的情况相同，骨架中的每一块骨骼都带动着与它关联的局部皮肤（在此，即骨骼所带动的众顶点）的形状与位置。如此一来，随着驱动骨骼，其附属的皮肤也会根据骨架的当前姿态相应地进行调整。

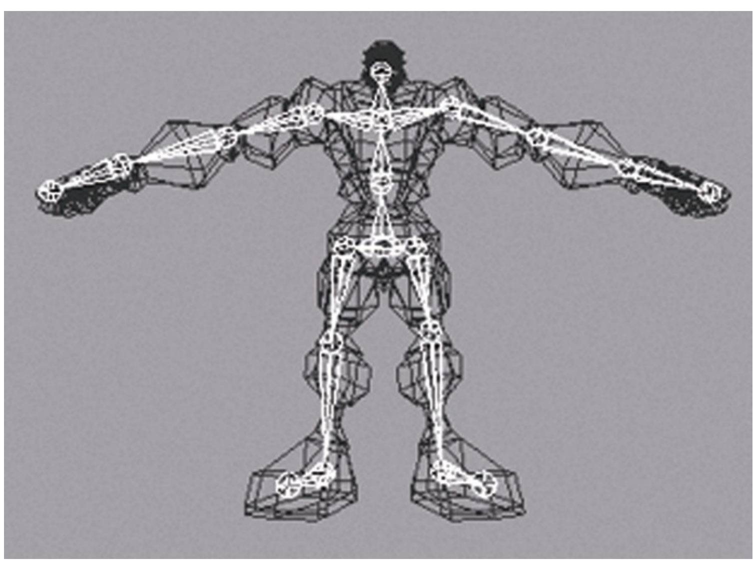


图23.6　一款角色网格。高亮标出的骨骼链表示角色的骨架，黑色多边形则代表人物的皮肤。

定义皮肤顶点所用的绑定空间（bind space）是在对整个皮肤网格建模时所参照的坐标系

23.2.2　重新推导将骨骼变换至根坐标系的公式

我们在本节中将推导的内容与23.1节有部分差别。第一点区别是：在这一节中，我们要把将对象从根坐标系变换至世界坐标系作为一个单独的步骤。因此，本节所求的并非是前文中将每块骨骼对象变换至世界空间的矩阵，而是要找寻把每块骨骼对象变换至根坐标系（to-root，即将骨骼对象从其局部坐标系转换至根坐标系所需的变换）的矩阵。

第二点不同之处在于，在23.1节中，我们是以自底向上的方式遍历目标对象的祖先节点，也就是说，以一块骨骼为起点上移至其祖先节点。但是，若按自顶向下的方式进行遍历，即以根节点开始向树状结构的下端移动其实效率更高（参见式（23.2））。用整数$0, 1, \ldots, n-1$对n块骨骼进行标记，我们就能以下列公式来表示第i块骨骼至根坐标系的变换：

$$\text{toRoot}_i = \text{toParent}_i \cdot \text{toRoot}_p \qquad (23.2)$$

其中，p为骨骼i的父对象的下标。这个公式究竟是什么意思呢？事实上，toRoot_p给出的是将几何体从骨骼p的坐标系直接快递至根坐标系的映射。所以，为了使几何体变换至根坐标系，我们仅需从骨骼i的坐标系中获取几何体，再通过toParent_i将几何体变换至骨骼i父对象p的坐标系即可。

现在整个流程中存在的唯一问题就是，在处理第i块骨骼的时候，

我们必须计算其父对象至根坐标系的变换（toRootp）。如果按自顶向下的顺序遍历树状结构，那么计算父对象“至根坐标系变换”的工作就总要先于计算其子对象的“至根坐标系变换”。

此时，“自顶向下”这种遍历方法便凸显出了更为高效的优势。若采用自顶向下的方式进行遍历，对于任意骨骼i来讲，我们事实上已经提前得到了将其父对象转换至根坐标系的变换矩阵。这样一来，我们离将骨骼i变换到根坐标系之中的目标只有一步之遥。而采用“自底向上”的遍历方法时，我们不仅要为每块骨骼遍历其全部的祖先，而且要在不同骨骼分享共用祖先对象这一情况下，多次进行重复的矩阵乘法。

23.2.3 偏移变换

事实上，角色皮肤的定义存在着一条微妙的细节，那就是受骨骼带动的顶点并非相对于此骨骼的坐标系而定义（它们的坐标都相对于绑定空间（bind space），即蒙皮网格建模所用的坐标系）。所以，在运用式（23.2）之前，我们首先需要将各皮肤顶点由绑定空间变换至牵动它们的骨骼的空间。这便是所谓的偏移变换（offset transformation），如图23.7所示。

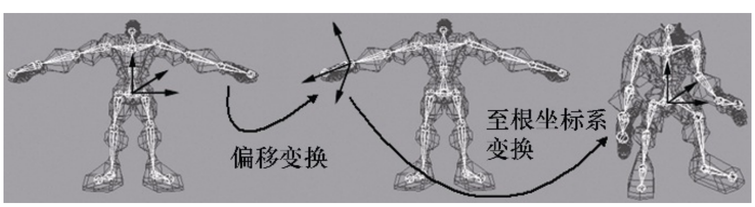


图23.7　首先通过偏移变换将受骨骼带动的皮肤顶点从绑定空间变换至此骨骼的空间。接着，待这些顶点位于骨骼空间后，就应用骨骼的“至根坐标系变换”将它们从骨骼空间转换到根空间

之中。最终变换（final tranoformation）即是把偏移变换与“至根坐标系变换”按顺序组合起来的变换

如此一来，我们就能通过任意骨骼B的偏移矩阵对相关皮肤顶点进行变换，将这些顶点从绑定空间转换至骨骼B的局部空间。只要相关皮肤顶点位于骨骼B的局部空间，我们就能利用骨骼B的“至根坐标变换”，将它们转换到角色空间中当前动画姿态的对应位置。

我们现在来介绍一种名为最终变换（final transform）的变换新品，其实就是把骨骼的偏移变换及其“至根坐标系变换”组合起来。从数学上来讲，第i块骨骼的最终变换矩阵$\boldsymbol{F}_i$的定义为：

$$\boldsymbol{F}_i = \text{offset}_i \cdot \text{toRoot}_i \qquad (23.3)$$

23.2.4 驱动骨架运动

在前一章的演示程序中，我们展示了如何驱使一个单独的物体运动。我们定义了关键帧（key frame）来指定物体在某一时刻的位置、朝向以及缩放大小。动画（animation）则为一系列按时间先后顺序排列的关键帧，这便是动画的粗略定义。另外，我们还演示了如何在关键帧之间进行插值，以计算出在两个关键帧之间某些时刻的物体方位。现在，我们通过扩展之前的动画系统来驱动多块骨骼运动。这些提及的动画类都被定义在本章例程“Skinned Mesh（蒙皮网格）”中的SkinnedData.h/.cpp文件内。

使一副骨骼运动并不比驱动一个单独的物体复杂多少。其实，我们

完全可以把单独的物体看作一块骨骼，而一副骨架就是一组连接在一起的骨骼。在此，我们假设每一块骨骼都能相对独立地运动。所以，要驱动一副骨架运动，我们先要令每块骨骼在它所在的局部范围内合理地活动。当每一块骨骼都能在它自己的天地里活动之后，我们就要将其祖先对象的运动对它的影响考虑在内，并将它转换至根空间。

我们定义了一段由一系列动作（每一个动作都需要骨架中的所有骨骼参与）组成的动画片段（animation clip），以令骨架为一种动作而协同工作。比如，"行走""跑路""攻击""下蹲"以及"跳跃"都可以当作动画片段的内容。

```
///<摘要你好>
/// AnimationClip的动作实例可以是"行走""跑动""攻击""下蹲"
/// 一段AnimationClip需要利用BoneAnimation来协调每块骨骼实现动画片段
///</摘要再见>
struct AnimationClip
{
  // 该动画片段里所有骨骼中最早开始运动的起始时间
  float GetClipStartTime()const;

  // 此动画片段里所有骨骼中最晚停止运动的结束时间
  float GetClipEndTime()const;

  // 遍历该动画片段中的每个BoneAnimation，并据此对动画进行插值
  void Interpolate(float t, std::vector<XMFLOAT4X4>& boneTransforms)const;

  // 用来指挥每一块骨骼运动
  std::vector<BoneAnimation> BoneAnimations;
};
```

为了使应用程序中的角色表现出所有的预定动作，我们往往会为它制作多种动画片段。由于一种角色的所有动画片段都要靠同一副骨架来实现，因此这些动作所用的骨骼块数都是相同的（尽管在完成某些特定动作时可能有部分骨骼丝毫不动）。正因如此，我们就能通过一

个unordered_map数据结构实例来存储所有的动画片段，并根据动画片段的识别名称来引用它们：

```
std::unordered_map<std::string, AnimationClip> mAnimations;
AnimationClip& clip = mAnimations["attack"];
```

最后，正如之前所提到的，针对每一块骨骼，我们都需要利用偏移变换，将受到其影响的诸皮肤顶点从绑定空间变换至该骨骼空间。另外，我们还需要以某种方式来表示骨架的层次结构（这里用的是一个数组——具体实现可详见下一小节）。下面介绍存储骨架动画数据的最终变换数据结构。

```
class SkinnedData
{
public:

  UINT BoneCount()const;

  float GetClipStartTime(const std::string& clipName)const;
  float GetClipEndTime(const std::string& clipName)const;

  void Set(
    std::vector<int>& boneHierarchy,
    std::vector<DirectX::XMFLOAT4X4>& boneOffsets,
    std::unordered_map<std::string, AnimationClip>& animations);

  // 在实际的项目中，如果需要调用clipName这一动画片段中同一timePos时刻的动画多次，我们可能
  // 就会渴望将此最终变换的结果缓存下来
  void GetFinalTransforms(const std::string& clipName, float timePos,
    std::vector<DirectX::XMFLOAT4X4>& finalTransforms)const;

private:
  // 用于给出第i块骨骼的父对象索引
  std::vector<int> mBoneHierarchy;

  std::vector<DirectX::XMFLOAT4X4> mBoneOffsets;

  std::unordered_map<std::string, AnimationClip> mAnimations;
```

```
};
```

23.2.5 计算最终变换

一款角色的框架层次往往用类似于图23.2所示的树状结构来表示。我们采用整型数组来模拟这种层次结构，即用第i个数组元素给出第i块骨骼的父对象索引。同时，还可用索引i来指示当前动画片段中对应的第i个BoneAnimation以及第i个偏移变换。而没有父对象的根骨骼总是处于元素0的位置。因此，骨骼i的祖父对象的动画数据以及偏移变换就可以按以下方式获得：

```
int parentIndex = mBoneHierarchy[i];
int grandParentIndex = mBoneHierarchy[parentIndex];

XMFLOAT4X4 offset = mBoneOffsets[grandParentIndex];

AnimationClip& clip = mAnimations["attack"];
BoneAnimation& anim = clip.BoneAnimations[grandParentIndex];
```

也可以据此计算出每块骨骼的最终变换，如：

```
void SkinnedData::GetFinalTransforms(const std::string& clipName,
float timePos, std::vector<XMFLOAT4X4>& finalTransforms)const
{
  UINT numBones = mBoneOffsets.size();

  std::vector<XMFLOAT4X4> toParentTransforms(numBones);

  // 根据给定的时间点对此动画片段中的所有骨骼进行插值
  auto clip = mAnimations.find(clipName);
  clip->second.Interpolate(timePos, toParentTransforms);

  //
  // 遍历该层次结构并将所有的骨骼变换至根空间
  //
```

```
  std::vector<XMFLOAT4X4> toRootTransforms(numBones);

  // 根骨骼的索引为0。由于根骨骼没有父对象，因此它的至根坐标系变换即为其自身局部
骨骼空间的变换
  toRootTransforms[0] = toParentTransforms[0];

  // 现在来求出子对象们的至根坐标系变换
  for(UINT i = 1; i < numBones; ++i)
  {
    XMMATRIX toParent = XMLoadFloat4x4(&toParentTransforms[i]);

    int parentIndex = mBoneHierarchy[i];
    XMMATRIX parentToRoot = XMLoadFloat4x4(&toRootTransforms[parentIndex])
;

    XMMATRIX toRoot = XMMatrixMultiply(toParent, parentToRoot);

    XMStoreFloat4x4(&toRootTransforms[i], toRoot);
  }

  // 令骨骼的偏移变换乘以至根坐标系变换来求得最终变换
  for(UINT i = 0; i < numBones; ++i)
  {
    XMMATRIX offset = XMLoadFloat4x4(&mBoneOffsets[i]);
    XMMATRIX toRoot = XMLoadFloat4x4(&toRootTransforms[i]);
    XMStoreFloat4x4(&finalTransforms[i], XMMatrixMultiply(offset,
    toRoot));
  }
}
```

此时，在变换阶段还需实现一项重要的细节。在循环中遍历每块骨骼的时候，我们要获取当前骨骼父对象的至根坐标系变换：

```
int parentIndex = mBoneHierarchy[i];
XMMATRIX parentToRoot = XMLoadFloat4x4(&toRootTransforms[parentIndex]);
```

只有在循环中最先求出父骨骼的“至根坐标系变换”，才能保证这项工作正确而顺利地执行下去。事实上，如果我们能够确保数组中的每块子骨骼都在其父骨骼之后的这一顺序，即可满足上述条件。而且，例程已经依此生成了适当的数据。以下是某个角色模型层次结构数组中前10

块骨骼的数据片段：

```
ParentIndexOfBone0: -1
ParentIndexOfBone1: 0
ParentIndexOfBone2: 0
ParentIndexOfBone3: 2
ParentIndexOfBone4: 3
ParentIndexOfBone5: 4
ParentIndexOfBone6: 5
ParentIndexOfBone7: 6
ParentIndexOfBone8: 5
ParentIndexOfBone9: 8
```

因此，骨骼9的父对象为骨骼8、骨骼8的父对象为骨骼5、骨骼5的父对象为骨骼4、骨骼4的父对象为骨骼3、骨骼3的父对象为骨骼2，骨骼2的父对象则为根节点骨骼0。由此便可以看出，父骨骼在数组中的位置从未被其子骨骼超越。

23.3　顶点混合

我们刚刚展示了如何来令骨架运动。在本节中，我们将把注意力聚焦在驱动附着于骨架上的皮肤（即蒙皮）。本书实现此功能所采用的算法称为顶点混合（vertex blending）。

顶点混合算法的策略如下。在皮肤之下即为呈层次结构的骨骼系统，而蒙皮本身是一段连续的网格（这就是说，不能把蒙皮按骨骼模型相应地划分成若干部分后再分别驱动它们）。由于皮肤上的同一个顶点可能受到一块或多块骨骼影响，因此最终的蒙皮效果要根据牵动它的各块骨骼的最终变换的加权平均值来确定（这些权值由美工在创建模型时指定并将之保存在文件中）。这样一来，即可在关节处（这通常是令人头疼的重点灾区）实现平滑的渐变混合效果，以此令皮肤表现得富有弹性。效果如图23.8所示。

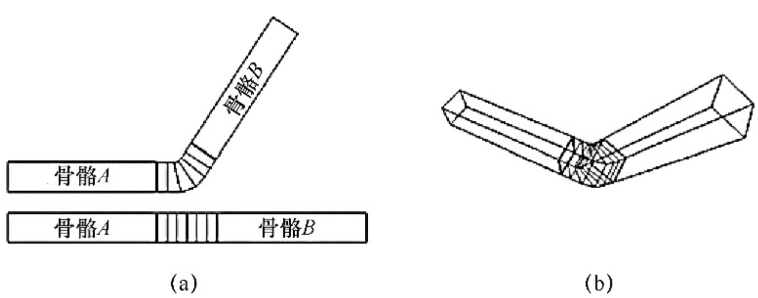


图23.8　皮肤是一块附着于骨骼的连续网格。由图中可以看出，关节附近的顶点被骨骼A与骨骼B同时牵动，以此为基础即可创建出平滑的渐变混合效果来模拟富有弹性的皮肤

[Möller08]提出：在实践的过程中，最好不要令同一顶点受到多于4块骨骼的影响。因此，在设计中就要考虑到每个顶点最多由4块骨骼来

驱动这一因素。这样，为了实现顶点混合技术，我们将会把角色的皮肤网格模拟为一块连续的网格。其中，每一个顶点都含有至多4个索引，以此对最终变换矩阵所构成的数组“骨骼矩阵调色板（bone matrix palette，骨架中的每块骨骼都与该数组中的元素一一对应）”进行索引。另外，每个顶点也最多有4个权值，分别用来描述每块骨骼对顶点影响的程度。据此，我们就得到了实现顶点混合所用的顶点结构体（见图23.9）。

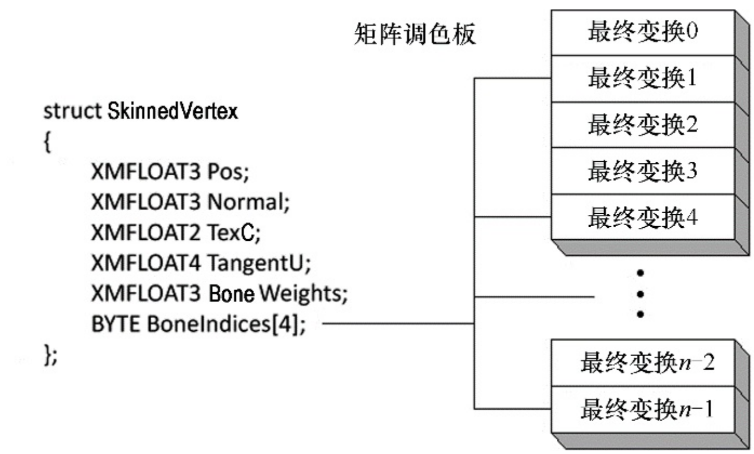


图23.9　矩阵调色板存储着每块骨骼的最终变换。从图中可以看出其中的4个骨骼索引是怎样在矩阵调色板中引用对应的最终变换的。同时，这些骨骼索引也反映出是骨架中的哪些骨骼在带动着不同的皮肤顶点。值得注意的是，顶点未必刚好被4块骨骼牵动。例如，4个索引里可能仅用到了两个，也就是说，此时只有两块骨骼在影响着目标顶点。我们也可以把某块骨骼的权值设置为0，从而去掉此骨骼对目标顶点的影响

如果为实现顶点混合而令一块连续网格中的顶点采用上述格式，我们就称此网格为蒙皮网格（skinned mesh）。

通过下列加权平均值公式，即可求出任何顶点$\boldsymbol{v}$相对于根标架的顶点混合位置（vertex-blended position）$\boldsymbol{v}'$（注意不要忘记，当所有的物

体都处于根坐标系中时，还需要在最后的步骤中进行世界变换，将它们变换至世界空间）：

$$\boldsymbol{v}' = w_0\boldsymbol{v}\boldsymbol{F}_0 + w_1\boldsymbol{v}\boldsymbol{F}_1 + w_2\boldsymbol{v}\boldsymbol{F}_2 + w_3\boldsymbol{v}\boldsymbol{F}_3$$

其中$w_0 + w_1 + w_2 + w_3 = 1$，即权值之和为1。

从公式中可以看出，我们在变换某个指定的顶点$\boldsymbol{v}$时，要按对其产生影响的骨骼最终变换（即矩阵$\boldsymbol{F}_0$、$\boldsymbol{F}_1$、$\boldsymbol{F}_2$、$\boldsymbol{F}_3$）单独进行计算。接着，再分别求出这些变换点的加权平均值，以计算出该顶点的最终混合位置$\boldsymbol{v}'$。

法线与切线的变换与之相似：

$$\begin{aligned}\boldsymbol{n}' &= \text{normalize}(w_0\boldsymbol{n}\boldsymbol{F}_0 + w_1\boldsymbol{n}\boldsymbol{F}_1 + w_2\boldsymbol{n}\boldsymbol{F}_2 + w_3\boldsymbol{n}\boldsymbol{F}_3)\\ \boldsymbol{t}' &= \text{normalize}(w_0\boldsymbol{t}\boldsymbol{F}_0 + w_1\boldsymbol{t}\boldsymbol{F}_1 + w_2\boldsymbol{t}\boldsymbol{F}_2 + w_3\boldsymbol{t}\boldsymbol{F}_3)\end{aligned}$$

这里我们假设变换矩阵$\boldsymbol{F}_i$中不含有任何非等比缩放变换。否则，使需要在对法线进行变换时，使用该矩阵的逆转置矩阵$(\boldsymbol{F}_i^{-1})^{\mathrm{T}}$（参见8.2.2节）。

下列顶点着色器片段展示了当每个顶点最多被4个骨骼影响时，实现顶点混合技术所用的关键代码：

```
cbuffer cbSkinned : register(b1)
{
  // 每个角色最多允许由96块骨骼构成
  float4x4 gBoneTransforms[96];
};

struct VertexIn
{
  float3 PosL   : POSITION;
```

```
  float3 NormalL : NORMAL;
  float2 TexC   : TEXCOORD;
  float3 TangentL : TANGENT;
#ifdef SKINNED
  float3 BoneWeights : WEIGHTS;
  uint4 BoneIndices : BONEINDICES;
#endif
};

struct VertexOut
{
  float4 PosH   : SV_POSITION;
  float4 ShadowPosH : POSITION0;
  float4 SsaoPosH   : POSITION1;
  float3 PosW   : POSITION2;
  float3 NormalW : NORMAL;
  float3 TangentW : TANGENT;
  float2 TexC   : TEXCOORD;
};

VertexOut VS(VertexIn vin)
{
  VertexOut vout = (VertexOut)0.0f;

  // 获取材质数据
  MaterialData matData = gMaterialData[gMaterialIndex];

#ifdef SKINNED
  float weights[4] = { 0.0f, 0.0f, 0.0f, 0.0f };
  weights[0] = vin.BoneWeights.x;
  weights[1] = vin.BoneWeights.y;
  weights[2] = vin.BoneWeights.z;
  weights[3] = 1.0f - weights[0] - weights[1] - weights[2];

  float3 posL = float3(0.0f, 0.0f, 0.0f);
  float3 normalL = float3(0.0f, 0.0f, 0.0f);
  float3 tangentL = float3(0.0f, 0.0f, 0.0f);
  for(int i = 0; i < 4; ++i)
  {
    // 假设在对法线进行变换时，采用的并不是非等比缩放，因此也就不必使用逆转置矩阵
了

    posL += weights[i] * mul(float4(vin.PosL, 1.0f),
         gBoneTransforms[vin.BoneIndices[i]]).xyz;
    normalL += weights[i] * mul(vin.NormalL,
           (float3x3)gBoneTransforms[vin.BoneIndices[i]]);
    tangentL += weights[i] * mul(vin.TangentL.xyz,
           (float3x3)gBoneTransforms[vin.BoneIndices[i]]);
```

```
  }

  vin.PosL = posL;
  vin.NormalL = normalL;
  vin.TangentL.xyz = tangentL;
#endif

  // 将顶点变换至世界空间
  float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
  vout.PosW = posW.xyz;

  // 假设执行的是均匀缩放，否则就需要使用世界矩阵的逆转置矩阵
  vout.NormalW = mul(vin.NormalL, (float3x3)gWorld);

  vout.TangentW = mul(vin.TangentL, (float3x3)gWorld);

  // 将顶点变换到齐次裁剪空间
  vout.PosH = mul(posW, gViewProj);

  // 为将SSAO图投射到场景之中而生成相应的投影纹理坐标
  vout.SsaoPosH = mul(posW, gViewProjTex);

  // 为三角形内插而输出顶点属性
  float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
  vout.TexC = mul(texC, matData.MatTransform).xy;

  // 为投影场景的阴影图而生成相应的投影纹理坐标
  vout.ShadowPosH = mul(posW, gShadowTransform);

  return vout;
}
```

如果上述顶点着色器是按“每个顶点最多被4个骨骼所牵动”来进行顶点混合的，那么为什么在处理每个顶点时仅输入3种权值而非4种权值呢？原因在于前面讲过的权值总和必为1。因此，要求出这第4个权值有$w_0 + w_1 + w_2 + w_3 = 1 \Leftrightarrow w_3 = 1 - w_0 - w_1 - w_2$。

23.4 从文件中加载动画数据

我们用文本文件来存储3D蒙皮网格以及动画数据，并称这种以.m3d作为扩展名的文件为“3D模型”文件。此文件格式是按照易加载和可读性强这两点来设计的，并非出于高性能这一目的，因而这种格式仅用于本书中的演示程序。

23.4.1 文件头

首先，.m3d格式定义的文件头指定了构成模型所需的材质、顶点、网格三角形、骨骼以及动画片段的数量。[4]

```
***************m3d-File-Header***************
#Materials 3
#Vertices 3121
#Triangles 4062
#Bones 44
#AnimationClips 15
```

1. **#Materials**：网格所用的不同材质数量。

2. **#Vertices**：网格使用的顶点数量。

3. **#Triangles**：构成网格的三角形数量。

4. **#Bones**：网格中骨骼的数量。

5. **#AnimationClips**：网格中动画片段的数量。

23.4.2　材质

.m3d格式中的下一个“区块”内容为一系列材质列表。下面的示例为soldier.m3d文件中的前两种材质。

```
***************Materials*********************
Name: soldier_head
Diffuse: 1 1 1
Fresnel0: 0.05 0.05 0.05
Roughness: 0.5
AlphaClip: 0
MaterialTypeName: Skinned
DiffuseMap: head_diff.dds
NormalMap: head_norm.dds

Name: soldier_jacket
Diffuse: 1 1 1
Fresnel0: 0.05 0.05 0.05
Roughness: 0.8
AlphaClip: 0
MaterialTypeName: Skinned
DiffuseMap: jacket_diff.dds
NormalMap: jacket_norm.dds
```

此文件中含有我们所熟知的材质数据（漫反射光、粗糙度等），但是也包括一些额外信息，如即将使用的纹理、是否要进行alpha裁剪以及材质类型名称。材质类型名称指出了哪些着色器程序需要使用当前所给定的材质。在上面的例子中，“Skinned”（蒙皮）类型说明了在支持蒙皮处理的着色器程序中会用到这种材质进行渲染。

23.4.3　子集

一整副网格是由一个或多个子集构成的。一个子集（subset）即为

网格中可以用同一种材质进行渲染的一组三角形。图23.10详细展示了如何把一个表示汽车的网格划分为若干子集。

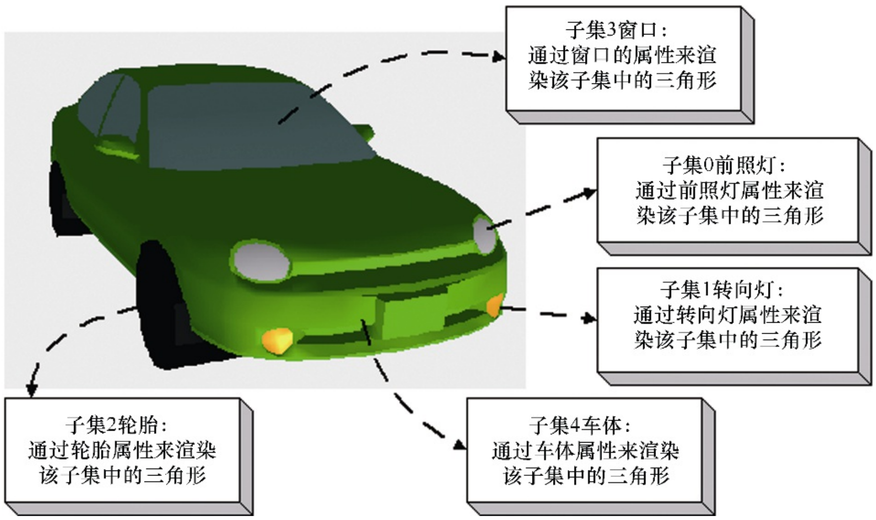


图23.10　一辆汽车的网格可以划分为若干子集。在此假设每个子集的区别仅在于材质。其实我们也可以想象得到，在实际应用中也可能存在对不同纹理运用加减混合的情况。另外，不同子集的渲染状态也可能各异。例如，玻璃窗可以用alpha混合的渲染方式来营造透明的效果。因此这些都可以作为划分子集的依据

子集与材质一一对应，即第i个子集对应于第i种材质。换句话说，第i个子集定义的是一段应由第i种材质来加以渲染的连续多个几何体图形。

```
***************SubsetTable******************
SubsetID: 0 VertexStart: 0 VertexCount: 3915 FaceStart: 0 FaceCount: 7230
SubsetID: 1 VertexStart: 3915 VertexCount: 2984 FaceStart: 7230 FaceCount:
 4449
SubsetID: 2 VertexStart: 6899 VertexCount: 4270 FaceStart: 11679 FaceCount
: 6579
SubsetID: 3 VertexStart: 11169 VertexCount: 2305 FaceStart: 18258 FaceCoun
t: 3807
SubsetID: 4 VertexStart: 13474 VertexCount: 274 FaceStart: 22065 FaceCount
: 442
```

在上面的示例中，第一个网格（即引用顶点[0, 3915)的网格）中的7230个三角形应当用材质0来渲染，而其后网格（即引用顶点[3915, 6899)的网格）中的4449个三角形应该以材质1来渲染。

23.4.4　顶点数据与三角形

接下来的两个数据区块只是罗列出了一系列的顶点与索引（每个三角形都有3个对应的索引）。

```
***************Vertices**********************
Position: -14.34667 90.44742 -12.08929
Tangent: -0.3069077 0.2750875 0.9111171 1
Normal: -0.3731041 -0.9154652 0.150721
Tex-Coords: 0.21795 0.105219
BlendWeights: 0.483457 0.483457 0.0194 0.013686
BlendIndices: 3 2 39 34

Position: -15.87868 94.60355 9.362272
Tangent: -0.3069076 0.2750875 0.9111172 1
Normal: -0.3731041 -0.9154652 0.150721
Tex-Coords: 0.278234 0.091931
BlendWeights: 0.4985979 0.4985979 0.002804151 0
BlendIndices: 39 2 3 0
...

***************Triangles*********************
0 1 2
3 4 5
6 7 8
9 10 11
12 13 14
...
```

23.4.5　骨骼偏移变换

骨骼偏移变换区块仅为每块骨骼都存储了一个4×4偏移变换矩阵。

```
***************BoneOffsets******************
BoneOffset0 -0.8669753 0.4982096 0.01187624 0
0.04897417 0.1088907 -0.9928461 0
-0.4959392 -0.8601914 -0.118805 0
-10.94755 -14.61919 90.63506 1

BoneOffset1 1 4.884964E-07 3.025227E-07 0
-3.145564E-07 2.163151E-07 -1 0
4.884964E-07 0.9999997 -9.59325E-08 0
3.284225 7.236738 1.556451 1
...
```

23.4.6 层次结构

层次结构区块存储的是层次结构数组，即一个整型数组，第i个数组元素保存的就是第i块骨骼的父索引。

```
***************BoneHierarchy****************
ParentIndexOfBone0: -1
ParentIndexOfBone1: 0
ParentIndexOfBone2: 1
ParentIndexOfBone3: 2
ParentIndexOfBone4: 3
ParentIndexOfBone5: 4
ParentIndexOfBone6: 5
ParentIndexOfBone7: 6
ParentIndexOfBone8: 7
ParentIndexOfBone9: 7
ParentIndexOfBone10: 7
ParentIndexOfBone11: 7
ParentIndexOfBone12: 6
ParentIndexOfBone13: 12
...
```

23.4.7 动画数据

最后一个区块所存的就是我们需要读取的动画片段。每一段动画都有一个识别名称以及针对骨架中每一块骨骼的一系列关键帧，而每一个关键帧又存储着时间点、用于指定骨骼位置的平移向量、用于指定骨骼大小的缩放向量以及指定骨骼朝向的四元数。

```
***************AnimationClips****************
AnimationClip run_loop
{
    Bone0 #Keyframes: 18
    {
        Time: 0 Pos: 2.538344 101.6727 -0.52932
        Scale: 1 1 1
        Quat: 0.4042651 0.3919331 -0.5853591 0.5833637
        Time: 0.0666666
        Pos: 0.81979 109.6893 -1.575387
        Scale: 0.9999998 0.9999998 0.9999998
        Quat: 0.4460441 0.3467651 -0.5356012 0.6276384
        ...
    }

    Bone1 #Keyframes: 18
    {
        Time: 0
        Pos: 36.48329 1.210869 92.7378
        Scale: 1 1 1
        Quat: 0.126642 0.1367731 0.69105 0.6983587
        Time: 0.0666666
        Pos: 36.30672 -2.835898 93.15854
        Scale: 1 1 1
        Quat: 0.1284061 0.1335271 0.6239273 0.7592083
        ...
    }
    ...
}

AnimationClip walk_loop
{
    Bone0 #Keyframes: 33
    {
        Time: 0
        Pos: 1.418595 98.13201 -0.051082
        Scale: 0.9999985 0.999999 0.9999991
        Quat: 0.3164562 0.6437552 -0.6428624 0.2686314
```

```
        Time: 0.0333333
        Pos: 0.956079 96.42985 -0.047988
        Scale: 0.9999999 0.9999999 0.9999999
        Quat: 0.3250651 0.6395872 -0.6386833 0.2781091
        ...
    }

    Bone1 #Keyframes: 33
    {
        Time: 0
        Pos: -5.831432 2.521564 93.75848
        Scale: 0.9999995 0.9999995 1
        Quat: -0.033817 -0.000631005 0.9097761 0.4137191
        Time: 0.0333333
        Pos: -5.688324 2.551427 93.71078
        Scale: 0.9999998 0.9999998 1
        Quat: -0.033202 -0.0006390021 0.903874 0.426508
        ...
    }
    ...
}

...
```

下列代码展示了从文件中读取动画片段数据的方法。

```
void M3DLoader::ReadAnimationClips(
  std::ifstream& fin,
  UINT numBones,
  UINT numAnimationClips,
  std::unordered_map<std::string,
  AnimationClip>& animations)
{
  std::string ignore;
  fin >> ignore; // AnimationClips的头部文本
  for(UINT clipIndex = 0; clipIndex < numAnimationClips; ++clipIndex)
  {
    std::string clipName;
    fin >> ignore >> clipName;
    fin >> ignore; // {

    AnimationClip clip;
    clip.BoneAnimations.resize(numBones);

    for(UINT boneIndex = 0; boneIndex < numBones; ++boneIndex)
    {
```

```
      ReadBoneKeyframes(fin, numBones, clip.BoneAnimations[boneIndex]);
    }
    fin >> ignore; // }

    animations[clipName] = clip;
  }
}

void M3DLoader::ReadBoneKeyframes(
  std::ifstream& fin,
  UINT numBones,
  BoneAnimation& boneAnimation)
{
  std::string ignore;
  UINT numKeyframes = 0;
  fin >> ignore >> ignore >> numKeyframes;
  fin >> ignore; // {

  boneAnimation.Keyframes.resize(numKeyframes);
  for(UINT i = 0; i < numKeyframes; ++i)
  {
    float t  = 0.0f;
    XMFLOAT3 p(0.0f, 0.0f, 0.0f);
    XMFLOAT3 s(1.0f, 1.0f, 1.0f);
    XMFLOAT4 q(0.0f, 0.0f, 0.0f, 1.0f);
    fin >> ignore >> t;
    fin >> ignore >> p.x >> p.y >> p.z;
    fin >> ignore >> s.x >> s.y >> s.z;
    fin >> ignore >> q.x >> q.y >> q.z >> q.w;

    boneAnimation.Keyframes[i].TimePos    = t;
    boneAnimation.Keyframes[i].Translation = p;
    boneAnimation.Keyframes[i].Scale     = s;
    boneAnimation.Keyframes[i].RotationQuat = q;
  }

  fin >> ignore; // }
}
```

23.4.8 M3DLoader类

从.m3d文件中加载数据的代码都位于LoadM3D.h/.cpp文件中，尤其

是LoadM3d函数。

```
bool M3DLoader::LoadM3d(
  const std::string& filename,
  std::vector<SkinnedVertex>& vertices,
  std::vector<USHORT>& indices,
  std::vector<Subset>& subsets,
  std::vector<M3dMaterial>& mats,
  SkinnedData& skinInfo)
{
  std::ifstream fin(filename);

  UINT numMaterials = 0;
  UINT numVertices = 0;
  UINT numTriangles = 0;
  UINT numBones   = 0;
  UINT numAnimationClips = 0;

  std::string ignore;

  if( fin )
  {
    fin >> ignore; // 文件的头文本
    fin >> ignore >> numMaterials;
    fin >> ignore >> numVertices;
    fin >> ignore >> numTriangles;
    fin >> ignore >> numBones;
    fin >> ignore >> numAnimationClips;

    std::vector<XMFLOAT4X4> boneOffsets;
    std::vector<int> boneIndexToParentIndex;
    std::unordered_map<std::string, AnimationClip> animations;

    ReadMaterials(fin, numMaterials, mats);
    ReadSubsetTable(fin, numMaterials, subsets);
    ReadSkinnedVertices(fin, numVertices, vertices);
    ReadTriangles(fin, numTriangles, indices);
    ReadBoneOffsets(fin, numBones, boneOffsets);
    ReadBoneHierarchy(fin, numBones, boneIndexToParentIndex);
    ReadAnimationClips(fin, numBones, numAnimationClips, animations);

    skinInfo.Set(boneIndexToParentIndex, boneOffsets, animations);

    return true;
  }
  return false;
```

```
}
```

像ReadMaterials这样的辅助函数实则都是用std::ifstream函数直接对文本文件进行解析。我们把这些函数的具体细节留给读者在阅读源代码时进行研究。

23.5 角色动画演示程序

正如我们在蒙皮网格着色器代码中所看到的，骨骼的最终变换都存于一个常量缓冲区中，在顶点着色器进行动画变换时将用到它们。

```
cbuffer cbSkinned : register(b1)
{
  // 每个角色最多支持96块骨骼
  float4x4 gBoneTransforms[96];
};
```

因此，我们还需要为处理每一个蒙皮网格对象，而向帧资源之中的添加一个常量缓冲区如下。

```
struct SkinnedConstants
{
  DirectX::XMFLOAT4X4 BoneTransforms[96];
};

std::unique_ptr<UploadBuffer<SkinnedConstants>> SkinnedCB = nullptr;

SkinnedCB = std::make_unique<UploadBuffer<SkinnedConstants>>(
  device, skinnedObjectCount, true);
```

我们需要为每一个动画角色的实例都设置一个`SkinnedConstants`结构体。一个动画角色实例通常由多个渲染项（每种材质都对应一个渲染项）组成，但是同一个角色实例的所有渲染项可以共享同一个`SkinnedConstants`，这是因为支持某一蒙皮网格运动的背后使用的都是同一组骨骼系统。

为了呈现动画角色实例在特定时刻的动作，我们定义了下列结构

体。

```
struct SkinnedModelInstance
{
  SkinnedData* SkinnedInfo = nullptr;

  // 用于存储给定时间点的最终变换
  std::vector<DirectX::XMFLOAT4X4> FinalTransforms;

  // 当前的动画片段名称
  std::string ClipName;

  // 动画时间点
  float TimePos = 0.0f;

  // 每一帧都要调用此函数，以推进时间点的位置、基于当前的动画片段对每一块骨骼进行
插值，并为顶点
  // 着色器中的处理流程生成最后应用于动画效果的最终变换
  void UpdateSkinnedAnimation(float dt)
  {
    TimePos += dt;

    // 使动画循环播放
    if(TimePos > SkinnedInfo->GetClipEndTime(ClipName))
      TimePos = 0.0f;

    // 为当前的时间点计算最终变换
    SkinnedInfo->GetFinalTransforms(ClipName, TimePos,
    FinalTransforms);
  }
};
```

接下来，向渲染项结构体添加下列数据成员。

```
struct RenderItem
{
  [...]
  // 指向骨骼变换常量缓冲区的索引。仅用于与蒙皮相关的渲染项
  UINT SkinnedCBIndex = -1;

  // 指向与此渲染项相关联的动画实例的指针
  // 如果该渲染项不受蒙皮网格的驱动，就将其设为nullptr
  SkinnedModelInstance* SkinnedModelInst = nullptr;
  [...]
```

```
};
```

我们要在每一帧都对动画角色实例进行更新（在本章的演示程序中其实只有一个角色实例）。

```
void SkinnedMeshApp::UpdateSkinnedCBs(const GameTimer& gt)
{
  auto currSkinnedCB = mCurrFrameResource->SkinnedCB.get();

  // 仅有一个蒙皮模型需要驱动
  mSkinnedModelInst->UpdateSkinnedAnimation(gt.DeltaTime());

  SkinnedConstants skinnedConstants;
  std::copy(
    std::begin(mSkinnedModelInst->FinalTransforms),
    std::end(mSkinnedModelInst->FinalTransforms),
    &skinnedConstants.BoneTransforms[0]);

  currSkinnedCB->CopyData(0, skinnedConstants);
}
```

在我们绘制渲染项时，如果某个渲染项被蒙皮网格所驱动，那么为其绑定与之相关的骨骼最终变换。

```
if(ri->SkinnedModelInst != nullptr)
{
  D3D12_GPU_VIRTUAL_ADDRESS skinnedCBAddress =
    skinnedCB->GetGPUVirtualAddress() +
    ri->SkinnedCBIndex*skinnedCBByteSize;
  cmdList->SetGraphicsRootConstantBufferView(1, skinnedCBAddress);
}
else
{
  cmdList->SetGraphicsRootConstantBufferView(1, 0);
}
```

图23.11所示的是本章演示程序的一张效果图。原始的动画模型以及纹理都出自DirectX SDK，我们出于演示的目的把它们转换为.m3d格

式。这个示例模型仅有一段名为Take1的动画片段。



图23.11 “Skinned Mesh”（蒙皮网格）演示程序的效果

23.6 小结

1．我们希望把现实世界中的大多数物体分解成具有父子对象关系的多个组成部分，以此方式来将它们转换为计算机程序中的图形化模型。在这个模型中，子对象可以自己相对独立地运动，但是当其父对象运动时，它也将受其影响而随之运动。例如，坦克的炮塔可以相对于坦克自行旋转，但是它仍要随着车体的移动而移动。另一个经典的例子是骨架，骨骼必须随其依附骨骼的运动而运动。考虑一列火车上的游戏角色。角色可以在列车中相对独立地运动，但是他们也会随着列车的行驶而移动。这个例子说明游戏中对象的层次结构是可以动态改变的，并必须时时更新。即在角色进入列车之前，列车并不是该角色层次结构中的一员。一旦玩家控制角色进入了列车，那么列车即刻就成为了角色层次结构中的一部分（即角色继承了火车运动的变换，所以也有人把“frame hierarchies”译作“帧继承”）。

2．网格层次结构中的每一个对象都是相对于其自身的局部坐标系来建模的，并根据位于其局部坐标系原点处的枢纽关节进行旋转。由于这些坐标系都存在于同一个世界空间之中，因此就能表示出它们之间的相对关系，继而通过变换把某个对象从它的局部空间变换到另一个对象的局部空间之中。特别是，我们能够以此来描述每个对象的坐标系相对于其父对象坐标系的关系。据此，我们就可以通过构建至父坐标系（to-parent）的变换矩阵，将一个对象的几何体从它的局部坐标系变换至其父坐标系。一旦该对象位于其父坐标系之中，我们就能通过其父对

象的“至父坐标系矩阵”再将此对象变换到其祖父对象的坐标系内，并依此类推，连续遍历每一个祖先对象的坐标系，直到最终抵达世界空间。换言之，为了将网格层次结构中的一个对象从其局部空间变换至世界空间，我们就要利用此对象乃至其所有祖先对象的“至父坐标系变换”（按升序的次序）沿坐标系层次结构上滤，直至该对象到达世界空间。如此一来，目标对象就会继承其祖先对象的变换，并随着其祖先对象的运动而运动。

3．我们可以用递归关系$\text{toRoot}_i = \text{toParent}_i \cdot \text{toRoot}_p$来表示第$i$块骨骼的至根坐标系变换。其中，$p$代表第$i$块骨骼的父骨骼。

4．骨骼偏移变换能够将各皮肤顶点从绑定空间变换到对它们产生影响相应骨骼的空间。骨架中的每一块骨骼都有一个与之对应的偏移变换。

5．在实现顶点混合技术的过程中，使用的蒙皮是一块连续的网格，皮肤的下面是呈层次结构的骨骼系统，因此蒙皮上的顶点可能受到一块或一块以上骨骼的牵扯。由于骨骼对顶点的影响程度取决于骨骼自身的权值。因此，针对4块骨骼影响同一个顶点的情况，我们可用加权平均公式$\boldsymbol{v}' = w_0\boldsymbol{v}\boldsymbol{F}_0 + w_1\boldsymbol{v}\boldsymbol{F}_1 + w_2\boldsymbol{v}\boldsymbol{F}_2 + w_3\boldsymbol{v}\boldsymbol{F}_3$来求取变换至骨架根坐标系的顶点$\boldsymbol{v}'$，其中$w_0 + w_1 + w_2 + w_3 = 1$。基于所用的连续网格以及每一个顶点都被若干加权骨骼所影响等因素，我们将获得一种更为自然的弹性蒙皮效果。

6．为了实现顶点混合技术，我们将每一块骨骼的最终变换矩阵都

存在了一个数组（这个数组称为矩阵调色板，matrix palette）之中（第i块骨骼的最终变换被定义为$\boldsymbol{F}_i = \text{offset}_i \cdot \text{toRoot}_i$，即该骨骼的偏移变换与其至根坐标系变换的乘积）。接下来，我们还针对每一个顶点保存了一系列顶点权重以及矩阵调色板索引。而一个顶点的矩阵调色板索引则用于找出影响着此顶点的所有骨骼的最终变换。[5]

23.7　练习

1．亲手建立一个可以运动的线性层次结构模型并对它进行渲染。例如，可以用球体网格与圆柱体网格构建出一个简单的机器手臂模型：以球体模拟关节，把圆柱体作为臂膀。

2．亲自创建一个如图23.12所示的可运动树状层次结构模型，并对它进行渲染。同时，也可以像练习1中那样，再次使用球体与圆柱体进行建模。

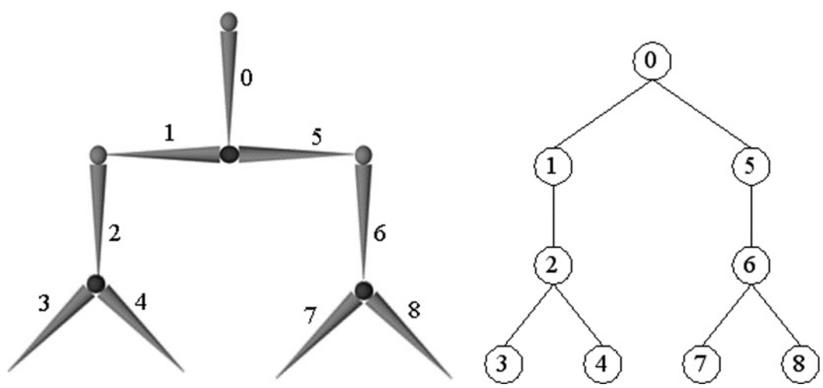


图23.12　一种简单的树状网格层次结构

3．如果读者手里有动画设计包（Blender是免费的），就可以尝试学习如何利用骨骼及其混合权值（blend weight）来建立一个简单的动画角色模型。随后，将生成的动画模型数据导出为支持顶点混合的.x格式文件，再试着把该文件数据转换为.m3d格式，以便在“Skinned Mesh”（蒙皮网格）演示程序中对该模型进行渲染。当然，这并不是一个短期内就能完成的项目。

[1]　原文为frame hierarchies，多译为“框架层次”。但译者认为翻译为“标架层次”也可行，甚至更妥，原因见下文。

[2]　有的文献中也把构成层次结构中的元素称为节点或子物体。

[3]　下文简记作父坐标系，类似的还有子坐标系、根坐标系等。

[4]　这里给出的都是示例数据，因此会与真正的源码有部分出入。

[5]　DirectX 12还有一些新的特性，本书并没有提到，比如“多GPU引擎以及多GPU适配器的使用与同步”（dn933254，后者提及得较少）。有关的官方示例为D3D12LinkedGpus与D3D12HeterogeneousMultiadapter。至于多线程方面，作者认为是高级话题，只是蜻蜓点水，也提到了要参考微软提供的D3D12Multithreading示例。事实上，多线程的有关函数本书基本都有涉及，关键在于实际应用上的绘制调度策略。有的读者可能希望了解DirectX 11程序在Windows 10上运行的方法。官网上有文专解移植问题《Porting from Direct3D 11 to Direct3D 12》（mt431709）。如果对程序执行效率的要求不太高，还可以从Direct3D 11on12这组API入手进行移植，有关的官方示例为D3D1211On12。本书作者在自己的官网d3dcoder上也给出了前一版DirectX 11程序在此环境中运行的方法《Direct3D 11 Book Demos with Windows 10 and Visual Studio 2015》。有网友提出例程在切换全屏模式时可能会崩溃，但他没说明环境。为保险起见，在此给出微软的相关示例D3D12Fullscreen。微软还推出了DirectX Raytracing（DXR）借此实现光线追踪（可见《Announcing Microsoft DirectX Raytracing!》）；并设计以WinML与DirectML在游戏

方面辅以机器学习（参见《Gaming with Windows ML》）。前文也提到过，随着Windows 10重要版本的更新，DirectX 12可能也会增加一些新特性，这可以参考New Releases（mt748631）。beyond3d论坛还有一个帖子《Direct3D feature levels discussion》，讨论是Direct3D 11与Direct3D 12每次升级的特性，并且还在持续更新中。对于初学者而言，找到苦海明灯进步似乎会比较快。有两篇此书前版的书评比较好，分别名为《我还是愿意尊它为龙书》与《龙书得失：请读 DirectX 9.0c A Shader Approach 版》。两文中都提到的Clayman的《游戏程序员养成计划》也推荐读一读。新兴论坛的崛起也亮出一批此方面的个中高手，比如Milo Yip、叛逆者以及空明流转等（不分先后且并不完整）。Milo Yip也曾列过一个与图形学学习有关的roadmap，名为《游戏程序员的学习之路（中英文两版）》（额，有网友美其名曰：游戏程序员的劝退之路。换句话说，最重要的是您要做什么？希望达到何种程度？）。另外，微软GitHub上的示例（DirectX-Graphics-Samples）一定要刷一遍。在提出与Direct3D 12相关的问题时，最好说明自己所用的具体环境，比如显卡型号、具体系统版本等，有时问题还可能与驱动有关！另外，本书在GitHub上的代码更新细节以及网友的讨论最好也看一看，例如有网友提到本书例程的性能还有利用其他参数来提升的空间等。

附录A　Windows编程入门

为了使用Direct3D应用程序编程接口（Application Programming Interface，API），我们先要创建出含有一个主窗口的Windows应用程序，并在此窗口中渲染3D场景。本附录的目标就是向读者介绍如何利用原生Win32 API来编写Windows应用程序。简单来讲，Win32 API是一套暴露给用户的底层函数与结构体，我们通过C语言调用它即可创建Windows应用程序。例如，要定义一个窗口类，我们就要填写Win32 API中的`WNDCLASS`结构体实例；要创建一个窗口，就要调用Win32 API的`CreateWindow`函数；要通知Windows系统呈现一个特定的窗口，就要使用Win32 API中的`ShowWindow`函数。

Windows编程是一门内容庞杂的主题，本附录仅介绍与我们所用的Direct3D紧密相关的部分知识。对以Win32 API编写Windows程序感兴趣读者，我们向您推荐查尔斯·佩措尔德（Charles Petzold）所著的Programming Windows（《Windows程序设计》）一书，目前，此书的最新版本为第5版[1]，它是此领域的参考标配。另一件无价之宝就是涵盖了所有微软技术的MSDN库，它通常被囊括在微软的Visual Studio之中，但也可以在线阅读。一般来说，如果我们偶尔遇到了一个比较陌生的Win32函数或结构体并希望对此了解得更多，就可以登录MSDN并搜索与该函数或结构体相关的所有信息。如果有读者感觉本附录中所谈及的某个Win32函数或结构体讨论得并不太深入，便可前往MSDN查看相

关的文档。

学习目标：

1．学习并理解Windows编程中常用到的事件驱动编程模型。

2．学习为使用Direct3D而创建Windows应用程序所需的最简代码。

注意 Note

为避免混淆，我们将使用大写字母“W”开头的Windows来表示视窗操作系统，而以小写字母“w”作为开头的window来代表Windows系统中的特定窗口。

A.1　概述

顾名思义，Windows编程的主题之一就是编写窗口程序。Windows应用程序由许多组件构成，如应用程序的主窗口、菜单、工具栏、滚动条、按钮以及其他的对话框控件。因此，一个Windows应用程序往往是由若干窗口构成的。后续小节会为Windows编程的诸多概念陆续给出简明的概述，在更全面的讨论开始之前，这些知识应当是我们所熟知的。

A.1.1　资源

在Windows系统中，多款应用程序可以并发运行。因此，像CPU、内存乃至显示器屏幕这些硬件资源，都在多应用程序的共享范围之列。为了防止多应用程序在无组织、无纪律的情况下访问或修改资源所引发的混乱，Windows应用程序并不能直接访问硬件。Windows系统的主要任务之一就是管理当前正在运行中的实例化程序并为它们合理分配资源。因此，为避免我们编写的程序因某些操作而对其他运行中的应用产生不必要的影响，这些具体的执行过程都要交由Windows系统来加以处理。例如，要展示一个窗口，我们必须调用Win32 API函数`ShowWindow`，而不能直接向显存中写入数据。

A.1.2　事件、消息队列、消息以及消息循环

凡是Windows应用程序就要依从事件驱动编程模型（event-driven

programming model）。一般来讲，应用程序总会“坐等”[2]某事的发生，即事件（event）的发生。生成事件的方式多种多样，常见的例子有键盘按键、点击鼠标，或者是窗口的创建、调整大小、移动、关闭、最小化、最大化乃至“隐身（visible，即窗口变为不可见的状态）”。

当事件发生时，Windows会向发生事件的应用程序发送相应的消息（message），随后，该消息会被添加至此应用程序的消息队列（message queue，简言之，这是一种为应用程序存储消息的优先级队列）之中。应用程序会在消息循环（message loop）中不断地检测队列中的消息，在接收到消息之后，它会将此消息分派到相应窗口的窗口过程（window procedure）。（一个应用程序可能附有若干个窗口。）每个窗口都有一个与之关联的名为窗口过程的函数[3]。我们实现的窗口过程函数中写有处理特定消息的代码。比如说，我们可能会希望在按下Esc键之后销毁窗口，此功能在窗口过程中可写作：

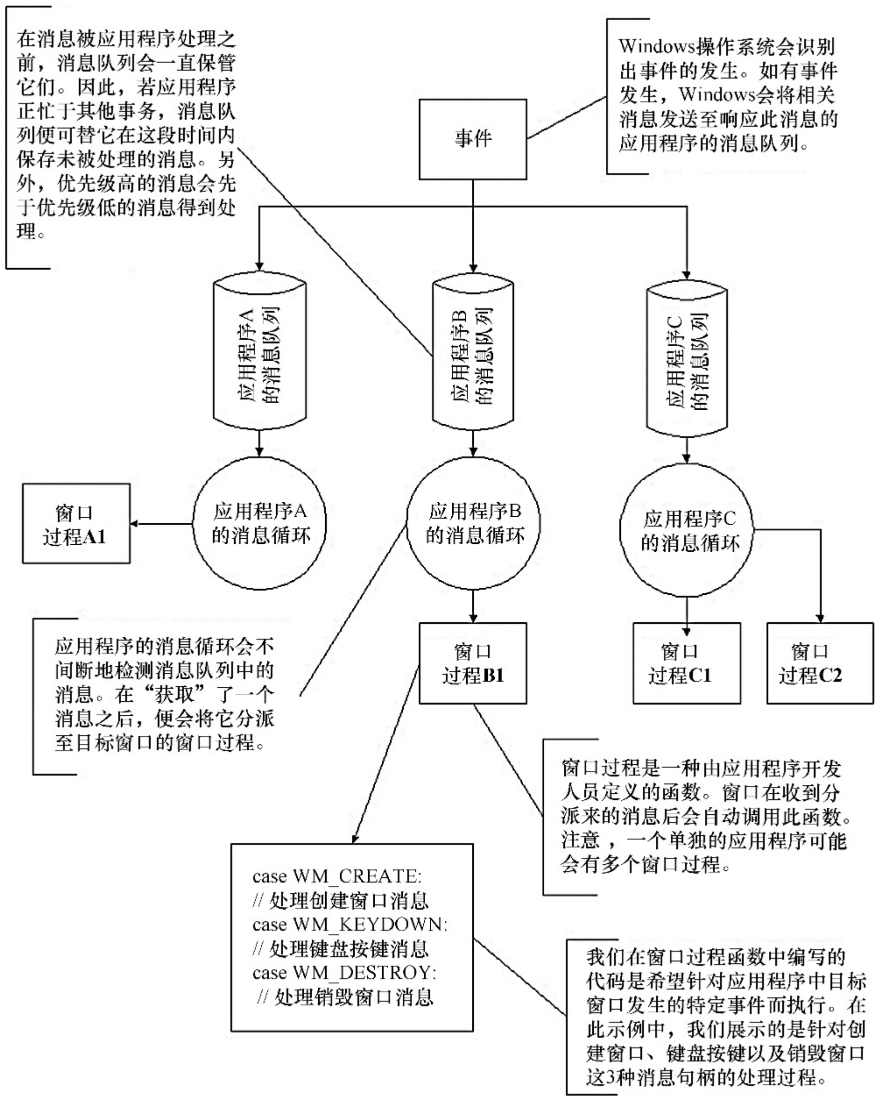
```
case WM_KEYDOWN:
        if( wParam == VK_ESCAPE )
                DestroyWindow(ghMainWnd);
        return 0;
```

我们应将目标窗口不处理的消息转发至Win32 API所提供的默认窗口过程`DefWindowProc`，让它去完成相应处理。

简而言之，用户或应用程序的某些行为会产生事件。操作系统会为响应此事件的应用程序发送相关的消息。随后，该消息会被添加到目标应用程序的消息队列之中。由于应用程序会不断地检测队列中的消息，在接收到消息后，应用程序就会将它分派到对应窗口的窗口过程。最

后，窗口过程会针对此消息执行相应的系列指令。

图A.1相对完整地概括了事件驱动编程的模型。

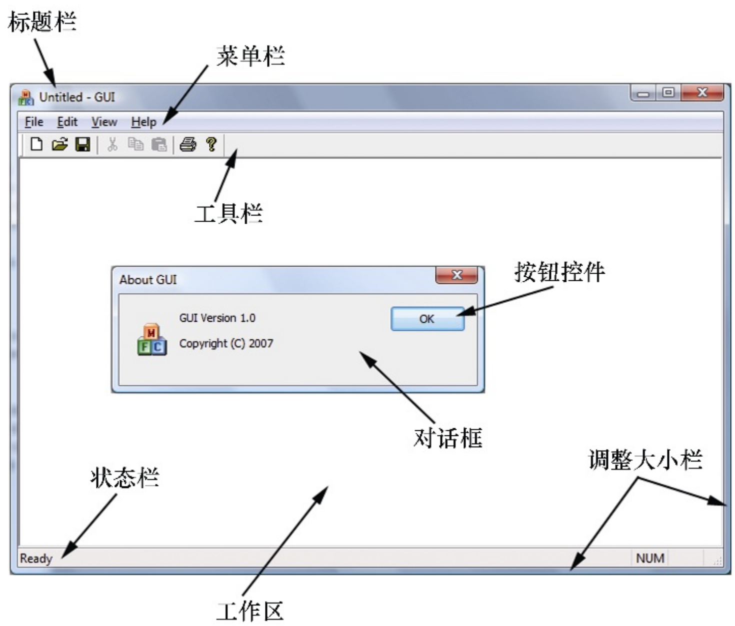


图A.1　编写Windows应用程序所用的事件驱动编程模型

A.1.3　图形用户界面

大多数的Windows程序会以图形用户界面（Graphical User

Interface，GUI）的表现形式呈现在用户面前并供其使用。典型的Windows应用程序应具有一个主窗口、一个菜单栏、一个工具栏，当然，或许还会有一些其他类型的控件。图A.2所示的是一些常见的GUI元素。对于Direct3D游戏编程来说，我们往往用不到过于复杂的GUI。事实上，在大多数情况下仅需一个主窗口即可，我们只是用它的工作区来渲染3D场景而已。



图A.2　典型的Windows应用程序GUI。此程序中最大的白色矩形范围即为工作区。一般来讲，这块区域即为大多数程序向用户展示输出内容的地方。在编写Direct3D应用程序的时候，我们就会将3D场景渲染到窗口的工作区中

A.1.4　Unicode

Unicode标准以16位值来表示一个字符。这样一来，我们就能通过它庞大的字符集来表示国际字符以及一些其他的符号。C++语言中以`wchar_t`类型的宽字符（wide-character）来表示Unicode码。不论是在

32位还是64位的Windows操作系统中，wchar_t都是16位的字符类型。在使用宽字符时，我们必须为字符串字面值（string literal）冠以大写字母前缀L，例如：

```
const wchar_t* wcstrPtr = L"Hello, World!";
```

前缀L会令编译器将字符串字面值作为宽字符串进行处理（即把char替换为wchar_t）。还有一个需要重视的问题是，我们在处理宽字符时还需使用相应版本的字符串函数。例如，在获取宽字符串的长度时，应使用wcslen函数而非strlen函数；在复制宽字符串时，应以wcscpy函数代替strcpy函数；在比较两个宽字符串时，该使用wcscmp函数而不是函数strcmp。这些宽字符版本的函数使用的也并不是char类型的指针，而是wchar_t类型的指针。不仅如此，C++标准库还专门提供了宽字符版本的字符串类std::wstring中。Windows头文件WinNT.h中亦有如下定义：

```
typedef wchar_t WCHAR; // wc, 16位的UNICODE字符
```

A.2 基本的Windows应用程序

以下代码描述的是一个麻雀虽小却五脏俱全的Windows程序。尽管在下一节中我们才会开始对这些代码进行讲解，但在此之前，读者最好先尽最大的努力按照给出的注释自行查阅一遍。另外，也建议您像对待习题那样，用开发工具创建一个工程，亲自输入这些代码，编译并执行。需要注意的是，我们在使用Visual C++开发与Direct3D有关的程序时，创建的必须是“Win32项目”（Win32 application project），而非“Win32控制台应用程序”（Win32 console application project）。

```
//=========================================================================
// Win32Basic.cpp的作者为Frank Luna (C) 2008版权所有
//
// 详述Direct3D 编程所需的Win32最简代码
//=========================================================================

// 包含windows头文件；其中含有编写Windows应用程序所需的所有Win32 API结构体、数
据类型以及
// 函数的声明
#include <windows.h>

// 用于指认所创建主窗口的句柄
HWND ghMainWnd = 0;

// 封装初始化Windows应用程序所需的代码。如果初始化成功，该函数返回true，否则返回
false
bool InitWindowsApp(HINSTANCE instanceHandle, int show);

// 封装消息循环代码
int Run();

// 窗口过程会处理窗口所接收到的消息
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);

// 在Windows应用程序中WinMain函数就相当于大多数语言中的main()函数
int WINAPI
```

```
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  PSTR pCmdLine, int nCmdShow)
{
    // 首先将hInstance 和nShowCmd 作为参数来调用封装函数（InitWindowsApp），用以创建和初始
    // 化应用程序的主窗口

        if(!InitWindowsApp(hInstance, nShowCmd))
        return 0;

    // 一旦将创建好的应用程序初始化完毕后，我们就可以开启消息循环了。随后，消息循环就会持续运转，
    // 直至接收到消息WM_QUIT——这表示此应用程序被关闭，该终止运行了
    return Run();
}

bool InitWindowsApp(HINSTANCE instanceHandle, int show)
{
    // 第一项任务便是通过填写WNDCLASS结构体，并根据其中描述的特征来创建一个窗口
    WNDCLASS wc;

    wc.style   = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance  = instanceHandle;
    wc.hIcon   = LoadIcon(0, IDI_APPLICATION);
    wc.hCursor  = LoadCursor(0, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = 0;
    wc.lpszClassName = L"BasicWndClass";

    // 下一步，我们要在Windows系统中为上述WNDCLASS注册一个实例，这样一来，即可据此创建窗口
    if(!RegisterClass(&wc))
    {
        MessageBox(0, L"RegisterClass FAILED", 0, 0);
        return false;
    }

    // 注册过WNDCLASS实例之后，我们就能用CreateWindow函数来创建一个窗口了。此函数返回的就
    // 是所建窗口的句柄（一个HWND类型的数值）。如果创建失败，返回的句柄数值为0。窗口句柄是一种
    // 窗口的引用方式， Windows系统会在内部对此进行管理。处理窗口的大量Win32 API函数都需要
    // 把HWND类型作为参数，以此来确定要对哪一个窗口执行相应的操作
```

```
    ghMainWnd = CreateWindow(
        L"BasicWndClass", // 创建此窗口采用的是前面注册的WNDCLASS实例
        L"Win32Basic",  // 窗口标题
        WS_OVERLAPPEDWINDOW, // 窗口的样式标志
        CW_USEDEFAULT,  // x坐标
        CW_USEDEFAULT,  // y坐标
        CW_USEDEFAULT,  // 窗口宽度
        CW_USEDEFAULT,  // 窗口高度
        0,     // 父窗口
        0,     // 菜单句柄
        instanceHandle,  // 应用程序实例句柄
        0);     // 可在此设置一些创建窗口所用的其他参数

    if(ghMainWnd == 0)
    {
        MessageBox(0, L"CreateWindow FAILED", 0, 0);
        return false;
    }

    // 尽管窗口已经创建完毕，但仍没有显示出来。因此，最后一步便是调用下面的两个函
数，将刚刚创建
    // 的窗口展示出来并对它进行更新。可以看出，我们为这两个函数都传入了窗口句柄，
这样一来，它们
    // 就知道需要展示以及更新的窗口是哪一个
    ShowWindow(ghMainWnd, show);
    UpdateWindow(ghMainWnd);

    return true;
  }

  int Run()
  {
      MSG msg = {0};

      // 在获取WM_QUIT消息之前，该函数会一直保持循环。GetMessage函数只有在收到W
M_QUIT 消
      // 息时才会返回0（false），这会造成循环终止；而若发生错误，它便会返回-1。
还需注意的一点
      // 是，在未有信息到来之时，GetMessage函数会令此应用程序线程进入休眠状态
      BOOL bRet = 1;
      while( (bRet = GetMessage(&msg, 0, 0, 0)) != 0 )
      {
          if(bRet == -1)
          {
                 MessageBox(0, L"GetMessage FAILED", L"Error", MB_OK);
                 break;
```

```
        }
        else
        {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
    }

    return (int)msg.wParam;
}

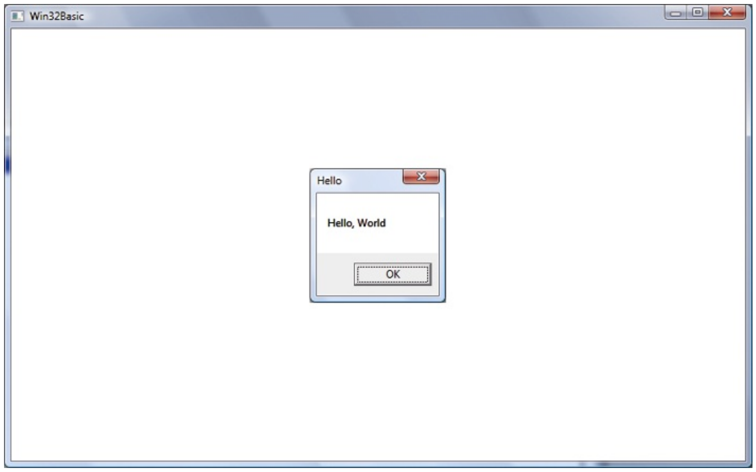
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    // 处理一些特定的消息。注意，在处理完一个消息之后，我们应当返回0
    switch( msg )
    {
      // 在按下鼠标左键后，弹出一个消息框
    case WM_LBUTTONDOWN:
      MessageBox(0, L"Hello, World", L"Hello", MB_OK);
      return 0;

      // 在按下Esc键后，销毁应用程序的主窗口
    case WM_KEYDOWN:
      if( wParam == VK_ESCAPE )
      DestroyWindow(ghMainWnd);
    return 0;

    // 处理销毁消息的方法是发送退出消息，这样一来便会终止消息循环
 case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
 }

 // 将上面没有处理的消息转发给默认的窗口过程。注意，我们自己所编写的窗口过程一
定要返回
 // DefWindowProc函数的返回值
return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

上述程序的执行效果如图A.3所示。



图A.3　上述程序的执行效果。图中演示的是当用户在窗口工作区按下鼠标左键时弹出的消息框。读者可以试一试按下Esc键，看看会发生什么

A.3　讲解基本Windows应用程序的工作流程

下面将对以上代码从头至尾考察一番，并根据调用的顺序逐步深入所用的一切函数。在阅读后续小节时，应对照上一节“基本的Windows应用程序”中所列的代码进行学习。

A.3.1　程序中的头文件、全局变量以及函数声明

我们在程序中要做的第一件事就是包含windows.h头文件。这样一来，我们就能够使用在windows.h文件中定义的各种结构体、数据类型以及函数声明这些Win32 API编程所需的基本元素。

```
#include <windows.h>
```

第二条语句是一个HWND类型的全局变量实例，它表示 “某个窗口的句柄”。在Windows编程中，我们通常采用Windows系统在内部为每个对象维护的句柄来处理相应的对象。在这个示例中，我们使用的就是Windows系统为应用程序主窗口维护的HWND句柄。保留该窗口句柄的原因是，有许多API需要针对特定窗口进行处理，因此参数中也就少不了窗口句柄的身影，它们会据此对相应的窗口执行函数的功能。例如，在调用UpdateWindow时就需要传入HWND类型的参数，该函数会对此句柄所引用的窗口进行更新。如果我们不向UpdateWindow函数传入句柄，它就无法知道要更新的窗口是哪一个。

```
HWND ghMainWnd = 0;
```

接下来的3行代码都是函数声明。简言之，InitWindowsApp函数创建并初始化应用程序的主窗口，Run函数封装了应用程序的消息循环，WndProc函数则是主窗口的窗口过程。我们会在调用这些函数的地方对它们的具体细节展开讨论。

```
bool InitWindowsApp(HINSTANCE instanceHandle, int show);
int Run();
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
```

A.3.2 WinMain

Windows编程中所用的WinMain函数就相当于在C++编程时通常用到的main函数。WinMain的原型如下。

```
int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    PSTR pCmdLine, int nCmdShow)
```

1. hInstance：当前应用程序的实例句柄。这是一种识别与引用当前应用程序的方式。前面曾提到多个Windows应用程序并发运行的情况，此时，通过句柄就可以便捷地引用所需的应用程序。

2. hPrevInstance：Win32编程用不到此参数，将其值设为0。

3. pCmdLine：运行此程序所用的命令行参数字符串。

4. nCmdShow：此参数指定了应用程序该如何显示。显示窗口的常

用命令有按窗口当前的大小与位置显示出来（SW_SHOW）、窗口最大化（SW_SHOWMAXIMIZED）以及窗口最小化（SW_SHOWMINIMIZED）。对于窗口显示命令的完整列表可参见MSDN库。

如果WinMain函数成功运行，那么在其终止时，它应当返回WM_QUIT消息的wParam成员（即退出值）。如果函数在退出时还未进入消息循环，那么它应该返回0。WINAPI标识符的定义为：

```
#define WINAPI  _stdcall
```

它指明了函数的调用约定，关乎函数参数的入栈顺序等。

A.3.3　WNDCLASS结构体与实例注册

在WinMain中我们调用了函数InitWindowsApp，借此完成了我们程序所需的所有初始化任务。现在我们来好好研究一下它的具体实现。InitWindowsApp函数返回的是一个布尔值，如果返回true则初始化成功，返回false则表示失败。在WinMain函数的实现中，我们将应用程序实例的副本以及窗口的显示命令变量都以参数的形式传入了InitWindowsApp函数，这两种数据都可从函数WinMain的参数列表中获取。

```
if(!InitWindowsApp(hInstance, nCmdShow))
```

初始化窗口的第一步就是通过填写WNDCLASS（window class，窗口类）结构体来描述窗口的基本属性。其定义如下：

```
typedef struct _WNDCLASS {
	UINT style;
	WNDPROC lpfnWndProc;
	int  cbClsExtra;
	int  cbWndExtra;
	HANDLE hInstance;
	HICON hIcon;
	HCURSOR hCursor;
	HBRUSH hbrBackground;
	LPCTSTR lpszMenuName;
	LPCTSTR lpszClassName;
} WNDCLASS;
```

1. style：指定了窗口类的样式。示例中使用的是CS_HREDRAW与CS_VREDRAW两种样式的组合。这两种位标志表示当工作区的宽度或高度发生改变时就重绘窗口。对于各种样式的完整描述可参考MSDN库。

```
wc.style = CS_HREDRAW | CS_VREDRAW;
```

2. lpfnWndProc：指向与此WNDCLASS实例相关联的窗口过程函数的指针。基于此WNDCLASS实例创建的窗口都会用到这个窗口过程。这就是说，若要创建两个采用同一窗口过程的窗口，仅需基于同一个WNDCLASS实例即可。如果希望以不同的窗口过程创建两个窗口，则需要为每个窗口都填写一个不同WNDCLASS实例。我们会在A.3.6节中讨论窗口过程函数。

```
wc.lpfnWndProc = WndProc;
```

3. cbClsExtra与cbWndExtra：我们可以根据需求，借助这两个字段来为当前应用分配额外的内存空间。我们现在编写的程序不需要这额外的空间，因此将它们统统设置为0。

```
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
```

4. hInstance：该字段是当前应用实例的句柄。前面曾提到，应用程序实例的句柄最早是通过WinMain函数传进来的。

```
wc.hInstance = instanceHandle;
```

5. hIcon：我们可以通过这个参数为以此窗口类创建的窗口指定一个图标的句柄。当然，我们可以使用自己设计的图标，但系统中也有一些内置的图标供我们选择，具体细节可参见MSDN库。下列语句采用的是默认的应用程序图标：

```
wc.hIcon = LoadIcon(0, IDI_APPLICATION);
```

6. hCursor：与hIcon相类似，我们可以借此指定在光标略过窗口工作区时所呈现的样式的句柄。同样，系统内置的光标资源也不少，详见MSDN库。下述代码采用的是标准的“箭头”光标。

```
wc.hCursor = LoadCursor(0, IDC_ARROW);
```

7. hbrBackground：该字段用来指出画刷（brush）的句柄，以此指定了窗口工作区的背景颜色。在此示例代码中，我们通过调用Win32函数GetStockObject返回了一个内置的白色画刷句柄。至于其他内置的画刷类型，可参考MSDN库。

```
wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
```

8. lpszMenuName：指定窗口的菜单。由于应用程序中没有菜单，

所以将它设为0。

```
wc.lpszMenuName = 0;
```

9. lpszClassName：指定所创窗口类结构体的名字。这个我们可以随意填写，我们在本示例中将它命名为“BasicWndClass”。有了这个名字，我们就可以在后续需要此窗口类结构体的时候方便地引用它。

```
wc.lpszClassName = L"BasicWndClass";
```

填写好一个WNDCLASS实例之后，为了使我们能够基于它来创建窗口还需要将它注册到Windows系统。通过RegisterClass函数就可实现这一点，它以指向欲注册的WNDCLASS结构体的指针作为参数，若注册失败则返回0。

```
if(!RegisterClass(&wc))
{
    MessageBox(0, L"RegisterClass FAILED", 0, 0);
    return false;
}
```

A.3.4　创建并显示窗口

在将一个WNDCLASS实例注册给Windows系统之后，我们就可以根据这个窗口类的描述来创建窗口了。通过我们赋予已注册WNDCLASS实例的名称（lpszClassName）便能对它进行引用。我们现在利用CreateWindow函数来创建窗口，下面是它的详细描述。

```
HWND CreateWindow(
```

```
        LPCTSTR lpClassName,
        LPCTSTR lpWindowName,
        DWORD dwStyle,
        int x,
        int y,
        int nWidth,
        int nHeight,
        HWND hWndParent,
        HMENU hMenu,
        HANDLE hInstance,
        LPVOID lpParam
);
```

1. lpClassName：存有我们欲创建窗口的属性的已注册WNDCLASS结构体的名。

2. lpWindowName：我们给窗口起的名称，它也将显示在窗口的标题栏中。

3. dwStyle：定义窗口的样式。WS_OVERLAPPEDWINDOW是由WS_OVERLAPPED（创建重叠窗口，一般具有标题栏和边框）、WS_CAPTION（具有一个标题栏的窗口）、WS_SYSMENU（标题栏中拥有系统菜单的窗口）、WS_THICKFRAME（使窗口具有可调整大小的边框）、WS_MINIMIZEBOX（具有最小化按钮的窗口）与WS_MAXIMIZEBOX（具有最大化按钮的窗口）六种标志组合而成的，从字面上就能看出它们所描述的窗口特征。窗口样式的完整列表可参见MSDN库。

4. x：窗口左上角的初始位置在于屏幕坐标系中的x坐标。我们可以将此参数指定为CW_USEDEFAULT，使Windows系统自动选择一个适当的默认值。

5．y：窗口左上角的初始位置在于屏幕坐标系中的y坐标。我们可以将此参数指定为CW_USEDEFAULT，使Windows系统自动选择一个适当的默认值。

6．nWidth：以像素为单位表示的窗口宽度。我们可以将其指定为CW_USEDEFAULT，Windowsx会自动选择适当的默认值。

7．nHeight：以像素为单位表示的窗口高度。我们可以将其指定为CW_USEDEFAULT，Windows会自动选择恰当的默认值。

8．hWndParent：所建窗口的父窗口句柄。由于我们创建的窗口不具有父窗口，因此将它设置为0。

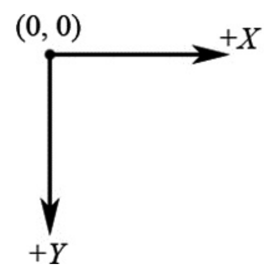
9．hMenu：菜单句柄。由于我们的程序无需菜单，因此将它设置为0。

10．hInstance：与此窗口相关联的应用程序句柄。

11．lpParam：一个指向用户定义数据的指针，可用作WM_CREATE消息的lpParam参数。在CreateWindow函数返回之前，会向待创建的窗口发送WM_CREATE消息。若要在窗口新建时执行某些操作（如初始化工作），则会处理WM_CREATE消息，而lpParam参数可传送处理过程中所用的数据。

注意 Note

我们指定的(x,y)坐标即窗口（左上角）相对于屏幕坐标系左上角（原点）的位置。在屏幕坐标系中，x轴的正方向依旧是水平向右的方向，而y轴的正方向则是垂直向下的方向。图A.4展示的正是这种坐标系，这被称为屏幕坐标系或屏幕空间。



图A.4　屏幕空间

CreateWindow函数返回的是它所创建窗口的句柄（类型为HWND）。如果窗口创建失败，则句柄的值为0（空句柄）。我们前面讲过，句柄是一种引用窗口的方式，它归于Windows系统管理。许多API的调用需要传入HWND，这样才能使函数找准要处理的窗口。

```
ghMainWnd = CreateWindow(L"BasicWndClass", L"Win32Basic",
       WS_OVERLAPPEDWINDOW,
       CW_USEDEFAULT, CW_USEDEFAULT,
       CW_USEDEFAULT, CW_USEDEFAULT,
       0, 0, instanceHandle, 0);
    if(ghMainWnd == 0)
    {
       MessageBox(0, L"CreateWindow FAILED", 0, 0);
       return false;
    }
```

最后要介绍的是在InitWindowsApp函数中，为显示窗口而必须调用的两种函数。这首先调用的是ShowWindow函数，我们向它传递新建

窗口的句柄，使它知道要显示的窗口是哪一个。除此之外，还要给它传入一个定义着窗口初次显示模式（例如最小化、最大化等）的整数值，这个值应当是WinMain函数的参数之一，即nCmdShow。展示了窗口之后，我们还应对它刷新，执行UpdateWindow函数的目的就在于此；该函数的参数是欲更新窗口的句柄。

```
ShowWindow(ghMainWnd, show);
    UpdateWindow(ghMainWnd);
```

如果InitWindowsApp函数运行到这里，那么初始化工作就已经大功告成，我们再返回s来表示一切顺利。

A.3.5 消息循环

待初始化工作都完成之后，我们就可以开始着手程序的核心——消息循环。在我们所编写的基本Windows应用程序之中，消息循环被封装在一个名为Run的函数内。

```
int Run()
{
    MSG msg = {0};

    BOOL bRet = 1;
    while( (bRet = GetMessage(&msg, 0, 0, 0)) != 0 )
    {
        if(bRet == -1)
        {
               MessageBox(0, L"GetMessage FAILED", L"Error", MB_OK);
               break;
        }
        else
        {
               TranslateMessage(&msg);
```

```
                DispatchMessage(&msg);
        }
    }

    return (int)msg.wParam;
}
```

Run函数要做的第一件事就是为表示Windows消息的MSG类型创建一个名为msg的变量实例。该结构体的定义如下。

```
typedef struct tagMSG {
        HWND hwnd;
        UINT message;
        WPARAM wParam;
        LPARAM lParam;
        DWORD time;
        POINT pt;
} MSG;
```

1. hwnd：接收此消息的窗口过程所属窗口的句柄。

2. message：用来识别消息的预定义常量值（如WM_QUIT）。

3. wParam：与此消息相关的额外信息，具体意义取决于特定的消息。

4. lParam：与此消息相关的额外信息，具体意义取决于特定的消息。

5. time：消息被发出的时间。

6. pt：消息发出时，鼠标指针位于屏幕坐标系中的坐标(x, y)。

接下来，程序进入到消息循环部分。GetMessage函数会从消息队

列中检索消息，并根据截获的消息细节填写msg的参数。由于我们不对消息进行过滤，因此将GetMessage函数的剩余参数均设为0。如果GetMessage函数发生错误，它将返回-1。若接收到WM_QUIT消息，GetMessage函数将返回0，继而终止当前的消息循环。如果GetMessage函数返回其他值，那么将继续执行下面的TranslateMessage与DispatchMessage两个函数。TranslateMessage函数实现了键盘按键的转换，特别是将虚拟键消息转换为字符消息；DispatchMessage函数则会把消息分派给相应的窗口过程。

如果应用程序根据WM_QUIT消息顺利退出，则WinMain函数将返回WM_QUIT消息的参数wParam（即退出代码）。

A.3.6 窗口过程

前文曾提到，我们在窗口过程中编写的代码是针对窗口接收到的消息而进行相应的处理。在本章这个基本的Windows应用程序之中，我们将窗口过程函数命名为WndProc，它的原型如下。

```
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
```

此函数将返回一个LRESULT类型的值（它的定义是一个整数），表示该函数调用是否成功。CALLBACK标识符指明这是一个回调（callback）函数，意味着Windows系统会在此程序的代码空间之外调用

该函数。就像我们在这个程序的源代码中所看到的，我们从没有主动显式地调用过这个窗口过程，这是因为Windows系统会在需要处理消息的时候自动为我们调用此窗口过程。

窗口过程的函数签名共有4个参数。

1．hWnd：接收此消息的窗口的句柄。

2．msg：标识特定消息的预定值。例如，窗口的退出消息被定义为WM_QUIT。前缀WM表示“窗口消息”（Window Message）。预定义的窗口消息有上百种，具体可参考MSDN库。

3．wParam：与具体消息相关的额外信息。

4．lParam：与具体消息相关的额外信息。

我们编写的窗口过程会处理3种消息，分别是WM_LBUTTONDOWN、WM_KEYDOWN与WM_DESTROY。当用户在窗口的工作区点击鼠标左键时，便会发送一次WM_LBUTTONDOWN消息。当有非键盘键被按下时，就会向具有当前焦点的窗口发送WM_KEYDOWN消息。当窗口被销毁时，便会发送WM_DESTROY消息。

我们编写的处理代码也相当简单，当接收到WM_LBUTTONDOWN消息时就弹出一个打印着“Hello, World”字样的消息框：

```
case WM_LBUTTONDOWN:
	MessageBox(0, L"Hello, World", L"Hello", MB_OK);
	return 0;
```

当窗口收到WM_KEYDOWN消息时，我们就先检测用户按下的是否为Esc键。若果真如此，则通过DestroyWindow函数销毁应用程序主窗口。此时，传入窗口过程的wParam参数即为用户按下的特定键的虚拟键代码（virtual key code），我们可以认为它是特定键的标识符。Windows头文件含有一系列用于确定按键的虚拟键代码。例如，通过检测虚拟键代码常量VK_ESCAPE，便可知晓用户按下的是否为Esc键。

```
case WM_KEYDOWN:
	if( wParam == VK_ESCAPE )
		DestroyWindow(ghMainWnd);
	return 0;
```

上文曾提到，参数wParam与lParam都被用于指定特定消息的额外信息。对于WM_KEYDOWN消息来讲，wParam参数指示的是用户按下的虚拟键代码。MSDN库为每一种Windows消息都罗列出了对应的wParam与lParam参数信息。

当窗口被销毁时，我们会以PostQuitMessage函数（该函数会终止消息循环）发出WM_QUIT消息。

```
case WM_DESTROY:
	PostQuitMessage(0);
	return 0;
```

在窗口过程的结尾，我们会调用另一个名为DefWindowProc的函数，这个函数是默认的窗口过程。在本章的基本Windows应用程序之中，我们编写的窗口过程仅能处理3种消息，而窗口接收到的其他消息则都要交给DefWindowProc函数，由其中定义的默认方法来进行处理。

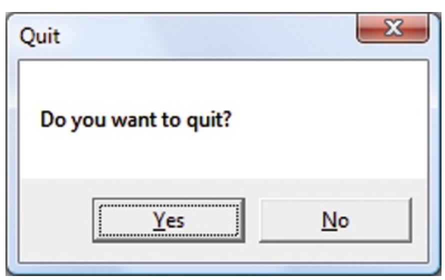
比如说，该程序的窗口可能需要执行最小化、最大化、调整大小或关闭等操作，由于我们并不希望自行处理这些消息，所以这些功能就要交由默认的窗口过程来实现。

A.3.7　消息框函数

最后，有一种API函数我们还未曾介绍过，那就是MessageBox函数。在向用户展示信息以及为程序快速地获取输入这两方面，它为我们提供了一种极其捷便的途径。消息框函数的声明如下：

```
int MessageBox(
 HWND hWnd,  // 该消息框所属窗口的句柄，可以指定为NULL
 LPCTSTR lpText, // 消息框中显示的文本
 LPCTSTR lpCaption,// 消息框的标题文本
 UINT uType  // 消息框的样式
);
```

MessageBox函数的返回值依赖于所用消息框的具体类型。对于可能的返回值与消息框样式，可参考MSDN库。图A.5所示的是一种带有“Yes”和“No”选项的消息框。



图A.5　带有“Yes”和“No”选项的消息框

A.4 一种更灵活的消息循环

对于办公软件或网络浏览器等传统应用程序而言，游戏软件与之差别较大。一般来讲，游戏程序采用的并非是坐等消息的模式，而是要时时进行更新。这便暴露出了一个问题，如果普通程序的消息队列中没有消息，那么函数GetMessage将使线程进入休眠状态并等待消息的到来。但与游戏程序相比，如果没有要处理的Windows消息就应执行游戏的逻辑代码。解决方法是以PeekMessage函数替代GetMessage函数。如果消息队列中并无消息，则PeekMessage函数将立即返回。这样一来，代码中的新式消息循环将变为：

```
int Run()
{
    MSG msg = {0};

    while(msg.message != WM_QUIT)
    {
        // 如果消息队列中有窗口消息则进行处理
        if(PeekMessage( &msg, 0, 0, 0, PM_REMOVE ))
        {
    TranslateMessage( &msg );
    DispatchMessage( &msg );
        }
        // 否则执行动画或游戏逻辑部分的代码
        else
    {

    }
    }
    return (int)msg.wParam;
}
```

实例化msg变量之后，我们将进入一个无限循环。在这里，首先要

调用API函数PeekMessage来检测消息队列，其参数的描述可参考MSDN库。若有消息，则返回true并对该消息进行处理。若没有消息，则PeekMessage函数返回false，然后执行我们编写的游戏逻辑代码。

A.5　小结

1．为了使用Direct3D，我们必须创建具有一个主窗口的Windows应用程序，以此来渲染3D场景。而且，对于游戏类程序而言，应创建一种用于检测消息的特殊消息循环。如果有消息则对它们进行处理，否则就执行游戏逻辑。

2．多个Windows应用程序可以同时运行，因此Windows操作系统必须管理这些程序所需资源，并将消息传递到相应的目标程序。当一个应用程序发生事件（键盘按键、点击鼠标、计时器等）时，就会有对应的消息发送至该应用程序的消息队列之中。

3．每个Windows应用程序都有一个消息队列，用于存储该程序接收到的消息。应用程序的消息循环会不断检测队列中的消息，并将它们分发到相应的目标窗口过程。值得注意是，一款应用程序可能会拥有多个窗口。

4．窗口过程是一种需要我们自行实现的特殊回调函数，当应用程序中的窗口收到消息，Windows操作系统便会立即调用它。在窗口过程的内部，我们根据自己的需求为特定消息的处理而编写执行代码。如果我们对某些消息没有特别的处理需求，则将它们转发到默认的窗口过程以默认方法进行处理。

A.6 练习

1．修改A.2节中的示例程序，为它配上不同的图标、光标以及背景颜色。

提示

在MSDN库中查阅`LoadIcon`、`LoadCursor`与`GetStockObject`这3种函数的信息。

2．修改A.2节中的示例程序，对`WM_CLOSE`消息进行相应的处理。接收到此消息的窗口或应用程序即将关闭。我们要实现的效果是：此消息发来时，通过消息框函数向用户显示出一个附有是/否选项的消息框，询问是否真的需要退出。如果用户选择“是”便销毁此窗口，否则返回之前的程序逻辑，继续正常运行。通过这种方法，我们就可以在程序退出时询问用户是否要保存当前的工作进度。

3．为A.2节中的示例程序添加对`WM_CREATE`消息的处理。在`CreateWindow`函数返回之前，此消息将被发送到即将被创建的窗口。通过消息框函数输出信息，以示此窗口创建完毕。

4．查阅MSDN库中的`Sleep`函数，并以自己的理解对此函数的功能进行总结。

5．查阅MSDN库中的`WM_SIZE`与`WM_ACTIVATE`消息，用自己的话来阐述这两种消息在什么情况下才会被发送出去。

---

[1]　此书的最新版本现为第6版。但针对本书Direct3D编程的这一主题，还是建议读者阅读此书的第5版。第5版使用C语言，内容介绍的是比较基础的Win32 API编程，与本书所需的知识更为接近。而第6版主要使用C#语言，更注重最新系统平台上的编程方法。

[2]　我们要注意的是应用程序可能会执行空闲处理（idle processing）。这是在没有事件发生时应用程序所执行的一种特殊任务。——原作者注

[3]　每个窗口都拥有一个窗口过程，但窗口之间也能共享相同的窗口过程。因此，不一定要为每个窗口都编写独立的窗口过程。当然，如果希望不同的窗口以不同的方式来处理消息，那么应为它们分别设置不同的窗口过程。——原作者注

附录B　高级着色器语言参考[1]

B.1　变量类型

B.1.1　标量类型

1. bool：取值非真即假。注意，HLSL为此而提供了类似于C++语言中的true与false关键字。

2. int：32位有符号整数。

3. half：16位浮点数。

4. float：32位浮点数。

5. double：64位浮点数。

有些平台可能不支持int、half与double这几种类型。遇到这种情况便会用float对这些类型进行模拟。

B.1.2　向量类型

1. float2：2D向量，其中的分量都是float类型。

2. **float3**：3D向量，其中的分量都是float类型。

3. **float4**：4D向量，其中的分量都是float类型。

注意 Note

我们也可以用非float类型的分量来创建向量，如int2、half3、bool4。

我们能够通过与数组或构造函数相似的语法来初始化向量。

```
float3 v = {1.0f, 2.0f, 3.0f};
float2 w = float2(x, y);
float4 u = float4(w, 3.0f, 4.0f); // u = (w.x, w.y, 3.0f, 4.0f)
```

还以利用数组下标语法来访问向量中的分量。比如说，要设置向量vec中的第i个分量可以写成：

```
vec[i] = 2.0f;
```

除此之外，我们还能够通过规定的分量名x、y、z、w、r、g、b、a像访问结构体成员那样来访问向量vec的各分量。

```
vec.x = vec.r = 1.0f;
vec.y = vec.g = 2.0f;
vec.z = vec.b = 3.0f;
vec.w = vec.a = 4.0f;
```

分量名r、g、b、a分别对应于分量名x、y、z、w，就像上述范例所示，每一对所获取的是同一个向量分量。当我们用向量来表示颜色时，RGBA表示法会更受青睐，这是因为它的分量名与表示的颜色更加吻合。

重组

考虑这样一种情况，假设有向量$\boldsymbol{u} = (u_x, u_y, u_z, u_w)$，欲将$\boldsymbol{u}$中的分量复制到向量$\boldsymbol{v}$，令$\boldsymbol{v} = (u_w, u_y, u_y, u_x)$。最直接的解决方案就是将向量$\boldsymbol{u}$中的分量根据需求分别复制到向量$\boldsymbol{v}$内。但是，HLSL为这种按乱序复制向量分量的情况，专门提供了一种名为重组（swizzleing，也有译作混合、重排等）的特殊语法。

```
float4 u = {1.0f, 2.0f, 3.0f, 4.0f};
float4 v = {0.0f, 0.0f, 5.0f, 6.0f};

v = u.wyyx; // v = {4.0f, 2.0f, 2.0f, 1.0f}
```

再如，

```
float4 u = {1.0f, 2.0f, 3.0f, 4.0f};
float4 v = {0.0f, 0.0f, 5.0f, 6.0f};

v = u.wzyx; // v = {4.0f, 3.0f, 2.0f, 1.0f}
```

这样，在复制向量时，就不必按部就班地依次复制每一个分量了。举个例子，我们可以如下列代码所示，仅将向量$\boldsymbol{u}$中的x、y分量复制到向量$\boldsymbol{v}$中。

```
float4 u = {1.0f, 2.0f, 3.0f, 4.0f};
float4 v = {0.0f, 0.0f, 5.0f, 6.0f};
```

```
v.xy = u; // v = {1.0f, 2.0f, 5.0f, 6.0f}
```

B.1.3　矩阵类型

我们能够以下列语法来定义一个$m \times n$矩阵，其中m与n的取值范围都在1～4。

```
floatmxn matmxn;
```

范例：

1. **float2x2**：2×2矩阵，该矩阵元素的类型都为**float**。
2. **float3x3**：3×3矩阵，该矩阵元素的类型都为**float**。
3. **float4x4**：4×4矩阵，该矩阵元素的类型都为**float**。
4. **float3x4**：3×4矩阵，该矩阵元素的类型都为**float**。

注意　Note

我们也可以用非**float**类型的分量来创建矩阵，如**int2x2**、**half3x3**和**bool4x4**。

我们能够以二维数组的双下标语法来访问矩阵中的项。例如，为了设置矩阵M中第i行第j列的元素，我们可以这样写：

```
M[i][j] = value;
```

除此之外，还可以依照访问结构体成员的方式来获取矩阵M中的元素。矩阵元素的名称定义如下：

以1作为起始值的索引：

```
M._11 = M._12 = M._13 = M._14 = 0.0f;
M._21 = M._22 = M._23 = M._24 = 0.0f;
M._31 = M._32 = M._33 = M._34 = 0.0f;
M._41 = M._42 = M._43 = M._44 = 0.0f;
```

以0作为基准值的索引：

```
M._m00 = M._m01 = M._m02 = M._m03 = 0.0f;
M._m10 = M._m11 = M._m12 = M._m13 = 0.0f;
M._m20 = M._m21 = M._m22 = M._m23 = 0.0f;
M._m30 = M._m31 = M._m32 = M._m33 = 0.0f;
```

有时候我们希望能提取引用矩阵中特定的行向量，这可以通过数组的单下标语法来实现。例如，为了提取3×3矩阵M中第i行的行向量，可写作：

```
float3 ithRow = M[i]; // 获取矩阵M中的第i个行向量
```

在接下来的示例中，我们要为矩阵中的3个行向量分别赋值：

```
float3 N = normalize(pIn.normalW);
float3 T = normalize(pIn.tangentW - dot(pIn.tangentW, N)*N);
float3 B = cross(N,T);
float3x3 TBN;
```

```
TBN[0] = T; // 给第一个行向量赋值
TBN[1] = B; // 给第二个行向量赋值
TBN[2] = N; // 给第三个行向量赋值
```

也可以用向量来构建矩阵：

```
float3 N = normalize(pIn.normalW);
float3 T = normalize(pIn.tangentW - dot(pIn.tangentW, N)*N);
float3 B = cross(N,T);

float3x3 TBN = float3x3(T, B, N);
```

注意 Note

除了利用float4与float4x4类型来表示4D向量与4×4矩阵外，我们还可以使用vector与matrix类型来加以代替：

```
vector u = {1.0f, 2.0f, 3.0f, 4.0f};
matrix M; // 4x4矩阵
```

B.1.4 数组

我们可以通过类似于C++的语法来声明一个特定类型的数组，例如：

```
float M[4][4];
half  p[4];
float3 v[12]; // 12个3D向量
```

B.1.5 结构体

HLSL与C++中定义结构体的方式基本上完全一致，区别仅在于HLSL中的结构体不具有成员函数。以下是一个HLSL结构体的相关示例。

```
struct SurfaceInfo
{
  float3 pos;
  float3 normal;
  float4 diffuse;
  float4 spec;
};

SurfaceInfo v;
litColor += v.diffuse;
dot(lightVec, v.normal);
float specPower  = max(v.spec.a, 1.0f);
```

B.1.6 typedef关键字

HLSL中的`typedef`关键字与C++中的功能完全相同。比如，可以通过以下语法来把`point`定义为`vector<float, 3>`类型的别称：

```
typedef float3 point;
```

此后，如果希望声明：

```
float3 myPoint;
```

就可以写作：

```
point myPoint;
```

再来看另一个示例，它展示了如何通过HLSL中的typedef与const关键字来定义类型（效果与C++中的一致）：

```
typedef const float CFLOAT;
```

B.1.7　变量的修饰符

下列关键字可以作为变量声明的修饰符。

1．static：基本上与修饰符extern的作用相反，意即以static修饰的着色器变量对于C++应用程序而言是不可见的。

```
static float3 v = {1.0f, 2.0f, 3.0f};
```

2．uniform：对于经此修饰符标记的变量而言，此值可以在C++应用层改变，但在着色器执行（处理顶点、像素）的过程中，其值始终保持不变（其间可将其看作一种常量）。因此，uniform变量都是在着色器程序之外进行初始化的（例如，在C++应用程序中得到初始化）。

3．extern：经此关键字修饰的变量会暴露给C++应用程序端（即该变量可以被着色器之外的C++应用代码访问到）。着色器程序中的全局变量被默认为既uniform且extern。

4．const：HLSL中的const关键字与C++中的意义相同。这就是说，如果一个变量由const关键字来修饰，那么它为一个常量，其值将不可更改。

```
const float pi = 3.14f;
```

B.1.8 强制类型转换

HLSL有着极其灵活的强制转换（casting）机制。HLSL中的强制转换语法与C编程语言中的相同。例如，将一个`float`类型强制转换为一个`matrix`类型可写作：

```
float f = 5.0f;
float4x4 m = (float4x4)f; // 将浮点数f复制到矩阵m中的每一元素当中
```

这种把标量强制转换为矩阵的做法，实为将此标量复制给矩阵中的每一元素。

再来研究下面的实例：

```
float3 n = float3(...);
float3 v = 2.0f*n - 1.0f;
```

`2.0f*n`只是一个标量与向量的乘法，这个好理解。然而，接下来的操作是向量与标量相减。此时，标量`1.0f`被扩充为`(1.0f, 1.0f, 1.0f)`，因此上述语句实际执行如下：

```
float3 v = 2.0f*n – float3(1.0f, 1.0f, 1.0f);
```

就本书中的示例而言，我们现在应当有能力根据语法推测出其强制转换后的语句。至于强制转换的一系列完整规则，可搜索SDK文档中的“Casting and Conversion”（强制类型转换与类型转换）部分。

B.2　关键字与运算符

B.2.1　关键字

为了便于参考，下面列出HLSL中定义的常用关键字清单[2]。

```
asm, bool, compile, const, decl, do,
double, else, extern, false, float, for,
half, if, in, inline, inout, int,
matrix, out, pass, pixelshader, return, sampler,
shared, static, string, struct, technique, texture,
true, typedef, uniform, vector, vertexshader, void,
volatile, while
```

接下来所示的关键字为保留的或未使用的标识符，但它们或许会在未来成为关键字[3]。

```
auto, break, case, catch, char, class,
const_cast, continue, default, delete, dynamic_cast, enum,
explicit, friend, goto, long, mutable, namespace,
new, operator, private, protected, public, register,
reinterpret_cast, short, signed, sizeof, static_cast, switch,
template, this, throw, try, typename, union,
unsigned, using, virtual
```

B.2.2　运算符

HLSL支持许多类似于C++中的运算符，除了后面介绍的少数几种特例之外，它们的用法与C++中的一致。下面表格中所列的即为HLSL中的运算符。

[]	.	>	<	<=	>=	!=	==	!
&&	‖	? :	+	+=	-	-=	*	*=
/	/=	%	%=	++	--	=	()	,

上面提到，HLSL中的运算符与C++中的非常相似，尽管如此，仍会有一些差异。首先，HLSL中的取模运算符%既可以用于整数也能用于浮点数。而且，要进行模运算就必须保证取模运算符左右操作数都具有相同的符号（例如，要么都为正数，要么都为负数）。

其次，我们可以看出HLSL中的不少运算是以分量为基准展开的，这是因为HLSL引入的向量和矩阵等内置类型都是由若干分量构成所导致的。有了这种按分量进行运算的规则，我们就可以将本用于标量类型运算的算符，亦用作执行如向量与矩阵之间的加、减法运算以及相等测试。下面是一些相关示例。

注意 Note

在标量计算的过程中，HLSL运算符的行为与一般普通C++程序中的相一致。

```
float4 u = {1.0f, 0.0f, -3.0f, 1.0f};
```

```
float4 v = {-4.0f, 2.0f, 1.0f, 0.0f};

// 将对应的分量相加
float4 sum = u + v; // sum = (-3.0f, 2.0f, -2.0f, 1.0f)
```

向量的自增运算就是使它的每个分量都加1。

```
// 自增之前：sum = (-3.0f, 2.0f, -2.0f, 1.0f)

sum++; // 自增之后：sum = (-2.0f, 3.0f, -1.0f, 2.0f)
```

向量的分量式乘法运算如下。

```
float4 u = {1.0f, 0.0f, -3.0f, 1.0f};
float4 v = {-4.0f, 2.0f, 1.0f, 0.0f};

// 将对应的向量相乘
float4 product = u * v; // product = (-4.0f, 0.0f, -3.0f, 0.0f)
```

注意 Note

如果有两个矩阵：

```
float4x4 A;
float4x4 B;
```

那么，语法$A * B$表示的是分量式乘法，而不是矩阵乘法。若要进行矩阵乘法运算，则需要使用`mul`函数。

比较运算符也是按分量一一进行比对的，它返回的是由`bool`类型分

量组成的向量或矩阵，其中的bool值即为对应分量的比较结果。例如：

```
float4 u = { 1.0f, 0.0f, -3.0f, 1.0f};
float4 v = {-4.0f, 0.0f, 1.0f, 1.0f};

float4 b = (u == v); // b = (false, true, false, true)
```

最后，我们以二元运算中变量类型的提升规则（variable promotion）作为本节的结束。

1．对于二元运算来说，如果运算符左右操作数的维度不同，那么维度较小的变量类型将被提升（自动转换）为维度较大的变量类型。比如说，如果x的类型为float，而y的类型为float3。那么，在表达式$(x+y)$中，变量x的类型将被提升为float3，而且表达的计算结果类型也为float3。变量类型的提升依赖于自动转换的定义，像本例这种情况就会发生“标量至向量”（scalar-to-vector）的自动转换。因此，在上述计算过程中，x就会按标量至向量自动转换的定义，提升为float3变为$x=(x,x,x)$。值得注意的是，如果某种自动转换行为没有定义，那么其相应的类型提升也就不复存在。例如，我们不能将类型float2提升为类型float3，因为并不存在这样的自动转换定义。

2．对于二元运算来讲，如果运算符左右操作数的类型不同，那么低精度变量的类型将被提升（自动转换）为高精度变量的类型。例如，若x的类型为int且y的类型为half，那么，在表达式$(x+y)$中，变量x的类型将被提升至half，而且表达式运算结果的类型亦为half。

B.3　程序中的控制流

HLSL支持一些C++中常见的选择、循环以及顺序程序流控制语句。这些语句的语法与C++中的一模一样。

*return*语句：

```
return (expression);
```

*if*与*if…else*语句：

```
if( condition )
{
  statement(s);
}

if( condition )
{
  statement(s);
}
else
{
  statement(s);
}
```

*for*语句：

```
for(initial; condition; increment)
{
  statement(s);
}
```

*while*语句：

```
while( condition )
```

```
{
  statement(s);
}
```

*do…while*语句：

```
do
{
  statement(s);
}while( condition );
```

B.4 函数

B.4.1 用户自定义函数

HLSL中的函数具有以下属性。

1．函数采用类C++语法。

2．参数只能按值传递。

3．不支持递归。

4．只有内联函数。

而且，HLSL还专为函数添加了一些额外的关键字。例如，考虑下列以HLSL编写的函数。

```
bool foo(in const bool b, // 输入的bool类型参数
     out int r1,    // 输出的int类型参数
     inout float r2) // 同时兼具输入/输出属性的float类型参数
{
  if( b ) // 测试输入值
  {
    r1 = 5; // 通过r1输出一个值
  }
  else
  {
    r1 = 1; // 通过r1输出一个值
  }

  // 因为r2的类型为inout，所以既可把它作为输入值，也能通过它来输出值
  r2 = r2 * r2 * r2;
```

```
  return true;
}
```

除了in、out与inout关键字以外，HLSL中函数的语法基本与C++中的函数相一致。

1．in：在目标函数执行之前，此修饰符所指定的参数应从调用此函数的程序中复制有输入的数据。我们其实不必显式地为参数指定此修饰符，在默认的情况下参数都为in类型。例如，下面两种写法是等价的：

```
float square(in float x)
{
  return x * x;
}
```

这是没有显式指定in的写法：

```
float square(float x)
{
  return x * x;
}
```

2．out：在目标函数返回时，此修饰符所指定的参数应当已经复制了该函数中的最后计算结果。借此即可方便地返回数值。关键字out的存在是很有必要的，因为HLSL既不允许按引用的方式传递数据，也不允许使用指针。需要注意的是，如果一个参数被标记为out，那么此参数在目标函数执行之前不能被复制任何值。换言之，out参数只能用于输出数据，而不能用于输入数据。

```
void square(in float x, out float y)
```

```
{
  y = x * x;
}
```

在这个函数中，我们通过x来输入底数，并以参数y来返回x平方的计算结果。

3. inout：此修饰符表示参数兼有in与out的属性。如果希望某个参数既可输入又可输出就可用inout。

```
void square(inout float x)
{
  x = x * x;
}
```

在这个示例中，我们不仅通过x来输入底数，还以x返回底数平方的计算结果。

B.4.2 内置函数

HLSL为用户提供了一组丰富的内置函数，这对于3D图形的绘制来讲无疑是很有帮助的。以下列表描述了常用的内置函数。

编号	函数	描述
1	abs(x)	返回$\|x\|$
2	ceil(x)	返回$\geqslant x$的最小整数

3	`cos(x)`	返回x的余弦值，x的单位为弧度
4	`clamp(x, a, b)`	将x钳制在范围$[a, b]$内，并返回结果
5	`clip(x)`	此函数仅限于在像素着色器中调用，如果$x < 0$，就将当前的像素从后续的处理流程中丢弃
6	`cross(u, v)`	返回$\boldsymbol{u} \times \boldsymbol{v}$（二向量之叉积）
7	`ddx(p)`	估算屏幕空间中的偏导数$\partial \boldsymbol{p} / \partial x$。这使我们可以确定在屏幕空间的$\boldsymbol{x}$轴方向上，相邻像素间某属性值$\boldsymbol{p}$的变化量
8	`ddy(p)`	估算屏幕空间中的偏导数$\partial \boldsymbol{p} / \partial y$。这使我们可以确定在屏幕空间的$\boldsymbol{y}$轴方向上，相邻像素间某属性值$\boldsymbol{p}$的变化量
9	`degrees(x)`	将x从弧度转化为角度
10	`determinant(M)`	返回矩阵的行列式
11	`distance(u, v)`	返回点$\boldsymbol{u}$与点$\boldsymbol{v}$之间的距离$\|\|\boldsymbol{v} - \boldsymbol{u}\|\|$
12	`dot(u, v)`	返回$\boldsymbol{u} \cdot \boldsymbol{v}$（二向量之点积）
13	`floor(x)`	返回$\leqslant x$的最大整数
		此函数返回的是目标浮点数的小数部分（即尾数）。例

14	`frac(x)`	如，若$x = (235.52, 696.32)$，那么` $\mathrm{frac}(x) = (0.52, 0.32)$
15	`length(v)`	返回$\|\boldsymbol{v}\|$（向量的模）
16	`lerp(u, v, t)`	基于参数$t \in [0,\ 1]$，在$\boldsymbol{u}$与$\boldsymbol{v}$之间进行线性插值
17	`log(x)`	返回$\ln(x)$
18	`log10(x)`	返回$\log_{10}(x)$
19	`log2(x)`	返回$\log_2(x)$
20	`max(x, y)`	如果$x \geqslant y$返回x，否则返回y
21	`min(x, y)`	如果$x \leqslant y$返回x，否则返回y
22	`mul(M, N)`	返回矩阵乘积$\boldsymbol{MN}$。但要注意，矩阵乘积$\boldsymbol{MN}$必须要有定义。如果$\boldsymbol{M}$是一个向量，就将它作为一个行向量进行向量与矩阵的乘法运算，使乘积有定义。类似地，如果$\boldsymbol{N}$是一个向量，就把它当作一个列向量，并进行矩阵与向量的乘法运算，使乘积有定义
23	`normalize(v)`	返回$\boldsymbol{v}/\|\boldsymbol{v}\|$
24	`pow(b, n)`	返回b^n

25	`radians(x)`	将x从角度转换为弧度
26	`saturate(x)`	返回`clamp`$(x, 0.0, 1.0)$
27	`sin(x)`	返回x的正弦值，x的单位为弧度
28	`sincos(in x, out s, out c)`	返回x的正弦值和余弦值，x的单位为弧度
29	`sqrt(x)`	返回$\sqrt{x}$
30	`reflect(v, n)`	根据给出的表面法线$\boldsymbol{n}$与入射向量$\boldsymbol{v}$，计算反射向量
31	`refract(v, n, eta)`	根据给出的表面法线$\boldsymbol{n}$、入射向量$\boldsymbol{v}$以及两种材质的折射率之比`eta`来计算折射向量
32	`rsqrt(x)`	返回$\frac{1}{\sqrt{x}}$
33	`tan(x)`	返回x的正切值，x的单位为弧度
34	`transpose(M)`	返回矩阵$\boldsymbol{M}$的转置矩阵$\boldsymbol{M}^{\mathrm{T}}$
35	`Texture2D::Sample (S, texC)`	根据`SamplerState`对象`S`与2D纹理坐标`texC`，返回2D纹理图中相应的颜色数据
		根据`SamplerState`对象`S`、2D纹理坐标`texC`以及mipmap层

36	Texture2D::SampleLevel(S, texC, mipLevel)	级mipLevel，从2D纹理图中返回相应的颜色数据。此函数与Texture2D::Sample函数的不同之处在于第三个参数，即需要用户亲自指定采样时所用的mipmap层级。例如，我们可以将它指定为0来访问最顶级的mipmap LOD（level of detail，细节级别，也有译作细节层次）
37	TextureCube::Sample(S, v)	根据SamplerState对象S以及3D查找向量*v*来返回立方体图中相应的颜色数据
38	Texture2DArray::Sample(S, texC)	根据SamplerState对象S（前文中曾讲过，采样器状态指定了纹理过滤器与纹理的寻址模式）以及3D纹理坐标texC（前两个坐标为普通的2D纹理坐标，而第三个坐标则用于指定数组的索引），返回2D纹理数组中相应的颜色

注意

大多数的函数能以重载的方式使所有内置类型的计算变得有意义。例如，所有的标量类型都可以通过abs函数来计算绝对值，这就是说，此函数会针对不同的标量类型执行不同的重载方法。再如叉积函数cross，它仅支持3D向量的计算，所以它只能针对由某种类型所组成的3D向量（例如，int、float、double等类型所构成的3D向量）进行重载。另一方面，因为线性插值函数lerp可以使标量、2D、3D以及4D

向量的计算均有意义，所以该函数支持对所有类型的重载。

注意 Note

如果以“标量”函数（即通常针对标量进行运算的函数，如`cos(x)`）来对非标量类型进行计算，那么该函数将对每个分量依次展开运算。例如，如果我们有下列代码：

```
float3 v = float3(0.0f, 0.0f, 0.0f);
v = cos(v);
```

那么`cos`函数将对每个分量逐个求取余弦值：$\boldsymbol{v} = (\cos(x), \cos(y), \cos(z))$。

如需进一步了解，可以在DirectX文档中的“HLSL Intrinsic Functions”（HLSL内置函数）部分找到内置函数的完整列表。

B.4.3　常量缓冲区的封装规则

在HLSL中，常量缓冲区会以补齐填充（padding）的方式，将其中的元素都包装为4D向量，但一个单独的元素却不能被分开并横跨两个

4D向量。思考下列示例：

```
// HLSL
cbuffer cb : register(b0)
{
  float3 Pos;
  float3 Dir;
};
```

如果要将数据打包为4D向量，则有读者可能会认为它是这样来实现的：

```
vector 1: (Pos.x, Pos.y, Pos.z, Dir.x)
vector 2: (Dir.y, Dir.z, empty, empty)
```

然而，这种方法却把dir元素分置在两个4D向量之中，这在HLSL的规则中是不允许的——一个元素只能置于一个4D向量之内。因此，它必须按下列方式封装在着色器的内存之中。

```
vector 1: (Pos.x, Pos.y, Pos.z, empty)
vector 2: (Dir.x, Dir.y, Dir.z, empty)
```

现假设我们用C++语言定义的即将被映射到常量缓冲区中的结构体如下。

```
// C++
struct Data
{
  XMFLOAT3 Pos;
  XMFLOAT3 Dir;
};
```

如果没有留意这些封装规则，而盲目地调用memcpy函数以复制字节的方式将数据写入常量缓冲区里，那么将以上面所述的第一种错误情

景而告终，而常量数据也将被错误地赋值为：

```
vector 1: (Pos.x, Pos.y, Pos.z, Dir.x)
vector 2: (Dir.y, Dir.z, empty, empty)
```

如此说来，我们必须依照HLSL的封装规则以“填充”变量的方式来定义C++结构体，以此使其中的数据元素可正确地复制给HLSL中的常量。现在，我们用显式填充的方法来重新定义常量缓冲区。

```
cbuffer cb : register(b0)
{
  float3 Pos;
  float __pad0;
  float3 Dir;
  float __pad1;
};
```

再定义C++代码中的结构体，以求与常量缓冲区相匹配。

```
// C++
struct Data
{
  XMFLOAT3 Pos;
  float __pad0;
  XMFLOAT3 Dir;
  float __pad1;
};
```

如果现在再调用memcpy函数，那么数据将被正确地复制到常量缓冲区之中。

```
vector 1: (Pos.x, Pos.y, Pos.z, __pad0)
vector 2: (Dir.x, Dir.y, Dir.z, __pad1)
```

为遵循封装规则，本书中的部分代码可能会采取填充变量的方法。

另外，在某些情况下，如果对常量缓冲区中的元素顺序进行适当调整，或许就能因减少占位的空白空间而避免填充操作。例如，若按下列方式定义Light结构体，我们就不必再填充4D向量的w坐标了——依此顺序排列的结构体元素使标量数据自然而然地占据了w坐标的位置（8.13.1节曾提及相关内容）。

```
struct Light
{
  DirectX::XMFLOAT3 Strength;
  float FalloffStart = 1.0f;
  DirectX::XMFLOAT3 Direction;
  float FalloffEnd = 10.0f;
  DirectX::XMFLOAT3 Position;
  float SpotPower = 64.0f;
};
```

当这些数据元素写入常量缓冲区时，它们会严丝合缝地打包为3个4D向量。

```
vector 1: (Strength.x, Strength.y, Strength.z, FalloffStart)
vector 2: (Direction.x, Direction.y, Direction.z, FalloffEnd)
vector 3: (Position.x, Position.y, Position.z, SpotPower)
```

注意 Note

最好按着色器内存中的常量缓冲区内存布局来定义C++常量缓冲区数据结构体，使两者匹配。届时，我们就可以无后顾之忧地进行内存复制操作。

刚刚摸清了HLSL包装与填充的路数，现在不妨就让我们多研究几种HLSL常量包装的实例。如果有以下常量缓冲区：

```
cbuffer cb : register(b0)
{
  float3 v;
  float  s;
  float2 p;
  float3 q;
};
```

那么，这一结构体将经过填充打包而成为下列3个4D向量：

```
vector 1: (v.x, v.y, v.z, s)
vector 2: (p.x, p.y, empty, empty)
vector 3: (q.x, q.y, q.z, empty)
```

在这种情况下，我们将标量s置为第一个向量中的第4个分量。然而，向量2的槽位还是不能同时容下变量p与q，因此q必须单独占用一个4D向量。

又如，考虑这样一个常量缓冲区：

```
cbuffer cb : register(b0)
{
    float2 u;
    float2 v;
    float a0;
    float a1;
    float a2;
};
```

它将被填充并封装为：

```
vector 1: (u.x, u.y, v.x, v.y)
vector 2: (a0, a1, a2, empty)
```

注意 Note

数组的处理方式与上文所述不同。根据SDK文档可知，“数组中的每个元素都会被存于一个具有4个分量的向量之中”。请看下面的示例，如果我们有一个类型为float2的数组：

```
float2 TexOffsets[8];
```

根据前文中的封装规则，我们可能会认为每两个元素会被包装到一个float4类型的槽位之中。但是，数组却是个例外，上述数组实际相当于：

```
float4 TexOffsets[8];
```

因此，为了令程序能按预期正常工作，我们需要在C++代码中设置一个具有8个XMFLOAT4元素的数组，而不是由8个XMFLOAT2元素所构成的数组。显而易见的是，每个元素都浪费了两个float类型的存储空间，毕竟我们其实只需要一个float2类型的数组。对此，SDK文档指出，我们可以通过强制类型转换以及额外的地址计算指令来更有效地利用内存：

```
float4 array[4];
static float2 aggressivePackArray[8] = (float2[8])array;
```

[1]　本章内容均为HLSL中的常用语法，便于快速查阅，完整细节请参考MSDN库。随着Direct3D的更新与升级， HLSL则有与之对应的着色器模型（shader model, SM），Direct3D 12的对应模型有SM 5.1与SM 6.x。

[2]　其中的关键字`decl`（与着色器汇编语言有关）已不在官方的HLSL列表之中，它仅残存在Directx 9的某些文献当中。

[3]　其中的`namespace`与`register`现已属于关键字，`break`、`continue`与`switch`则已归入流控制语句。

附录C　解析几何学选讲

在本附录中，我们将以向量与点为基石来构建出更为复杂的几何图形。本书中虽然经常会用到这些知识，但仍不如向量、矩阵以及变换这些内容出现得频繁。正因如此，我们才把这些主题放到附录中而非正文当中。

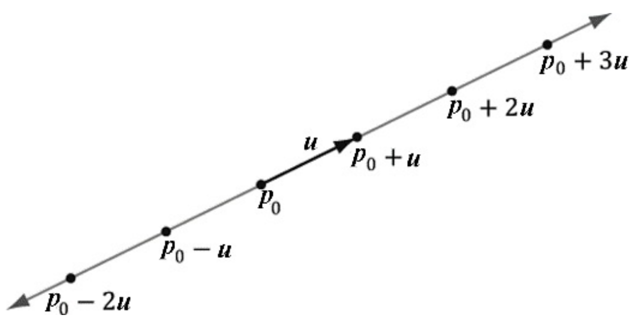
C.1　射线、直线以及线段

一条直线可以用线上一点$\boldsymbol{p}_0$以及与此线平行的向量$\boldsymbol{u}$来表示（见图C.1）。所以，此直线的向量直线方程（vector line equation）为：

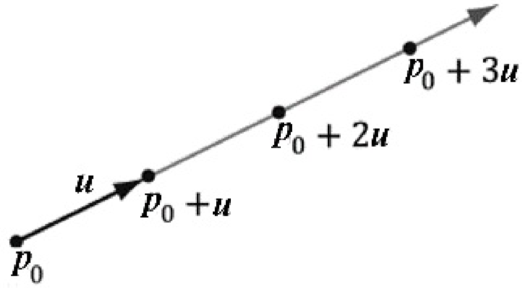
$$\boldsymbol{p}(t)=\boldsymbol{p}_0+t\boldsymbol{u}\qquad 其中t\in\mathbb{R}$$

通过赋予t（t可以是任意实数）不同的值，我们就能获得此直线上的不同点。

如果将t限制为一个非负数，那么这个向量直线方程的图像将变为以$\boldsymbol{p}_0$作为端点、向量$\boldsymbol{u}$表示方向的射线（见图C.2）。

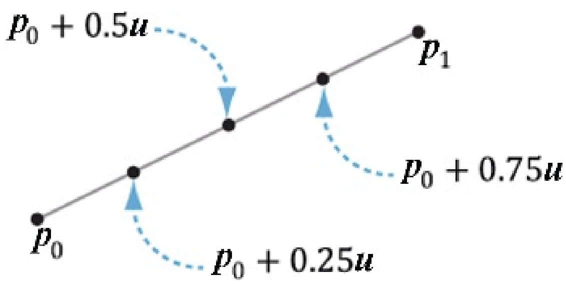


图C.1　由线上一点$\boldsymbol{p}_0$以及与此线平行的向量$\boldsymbol{u}$来描述的直线。我们可以通过为t设置任意实数来得到直线上的对应点



图C.2　由端点$\boldsymbol{p}_0$与方向向量$\boldsymbol{u}$所描述的射线。我们通过为t设置大于或等于0的标量来生成该射线上的对应点

现假设欲通过端点$\boldsymbol{p}_0$与$\boldsymbol{p}_1$来定义一条线段。首先要构建出由$\boldsymbol{p}_0$至$\boldsymbol{p}_1$的向量$\boldsymbol{u}=\boldsymbol{p}_1-\boldsymbol{p}_0$，如图C.3所示。接下来，针对$t\in[0,\ 1]$绘制出方程$\boldsymbol{p}(t)=\boldsymbol{p}_0+t\boldsymbol{u}=\boldsymbol{p}_0+t(\boldsymbol{p}_1-\boldsymbol{p}_0)$的图像，即为由$\boldsymbol{p}_0$与$\boldsymbol{p}_1$所定义的线段。注意，如果我们在绘制图像时没有满足$t\in[0,\ 1]$这个条件，那么我们所绘制的点将位于该线段所在的直线上，但不在此线段之上。

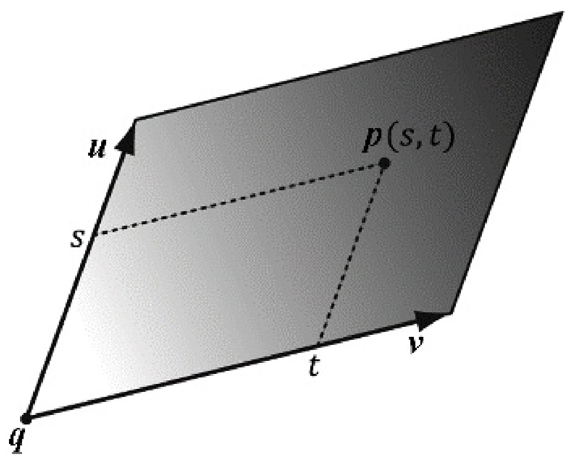


图C.3　通过为t赋予范围为[0, 1]的值来生成线段上的点。例如，当$t=0.5$时，便生成线段的中点。 观察图像可知，当$t=0$时，我们得到的是端点$\boldsymbol{p}_0$；当$t=1$，就可得到端点$\boldsymbol{p}_1$

C.2 平行四边形

设$\boldsymbol{q}$为一个点，且$\boldsymbol{u}$与$\boldsymbol{v}$为两个非平行向量（即对于任意标量k来讲，$\boldsymbol{u} \neq k\boldsymbol{v}$）。下列函数的图像即为一个平行四边形（见图C.4）。

$$\boldsymbol{p}(s,t) = \boldsymbol{q} + s\boldsymbol{u} + t\boldsymbol{v} \qquad \text{其中} s, t \in [0,\ 1]$$



图C.4　一个平行四边形。通过为$s, t \in [0,\ 1]$这两个变量取约束范围内的不同值，我们就能根据上述方程生成对应平行四边形中不同的点

需要满足“对于任意标量k都要使$\boldsymbol{u} \neq k\boldsymbol{v}$”这个条件的原因如下。

如果$\boldsymbol{u} = k\boldsymbol{v}$，那么就有：

$$\begin{aligned}\boldsymbol{p}(s,t) &= \boldsymbol{q} + s\boldsymbol{u} + t\boldsymbol{v} \\ &= \boldsymbol{q} + sk\boldsymbol{v} + t\boldsymbol{v} \\ &= \boldsymbol{q} + (sk+t)\boldsymbol{v} \\ &= \boldsymbol{q} + \bar{t}\boldsymbol{v}\end{aligned}$$

这其实就是直线方程。换句话说，我们现在得到的是一个单自由度（one degree of freedom，也作一自由度）方程。而要获得像平行四边形

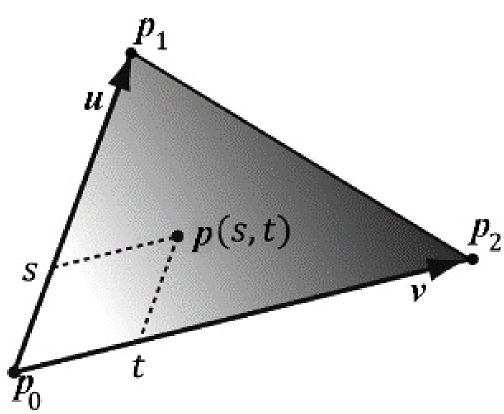
那样的2D几何形状，则必须采用双自由度（二自由度）方程。因此，$\boldsymbol{u}$与$\boldsymbol{v}$必不能互为平行向量。

C.3　三角形

三角形的向量方程与平行四边形的向量方程差不多，唯一区别就是前者更进一步限制了参数的范围：

$$\boldsymbol{p}(s,t)=\boldsymbol{p}_0+s\boldsymbol{u}+t\boldsymbol{v}\quad 其中\quad s\geqslant 0,t\geqslant 0,s+t\leqslant 1$$

从图C.5中可以看出，如果在与s、t有关的条件里有任何一项没有得到满足，那么$\boldsymbol{p}(s,t)$则为在三角形所在平面上，却处于三角形范围“之外”的一个点。



图C.5　一个三角形。通过为参数s、t取满足$s\geqslant 0,t\geqslant 0,s+t\leqslant 1$的不同值，我们就可以生成此三角形内的不同点

根据定义三角形的3个点，我们就能得到三角形的参数方程。现考虑由3个顶点$\boldsymbol{p}_0$、$\boldsymbol{p}_1$、$\boldsymbol{p}_2$所定义的三角形。我们可以把满足$s\geqslant 0,t\geqslant 0,s+t\leqslant 1$的位于三角形内的任意一点表示为：

$$\boldsymbol{p}(s,t)=\boldsymbol{p}_0+s(\boldsymbol{p}_1-\boldsymbol{p}_0)+t(\boldsymbol{p}_2-\boldsymbol{p}_0)$$

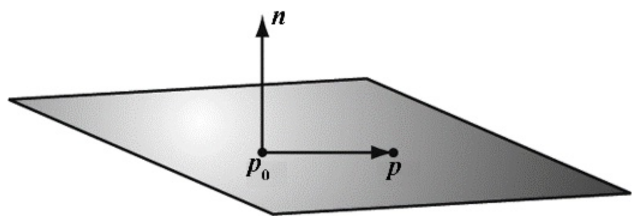
根据标量乘法对向量加法分配律，对公式进一步展开：

$$\begin{aligned}\boldsymbol{p}(s,t) &= \boldsymbol{p}_0 + s\boldsymbol{p}_1 - s\boldsymbol{p}_0 + t\boldsymbol{p}_2 - t\boldsymbol{p}_0 \\ &= 1 - s - t)\boldsymbol{p}_0 + s\boldsymbol{p}_1 + t\boldsymbol{p}_2 \\ &= r\boldsymbol{p}_0 + s\boldsymbol{p}_1 + t\boldsymbol{p}_2\end{aligned}$$

这里设$r = (1 - s - t)$。坐标(r, s, t)则称为重心坐标（barycentric coordinate）。注意，通过约束条件$r + s + t = 1$与重心组合$\boldsymbol{p}(r, s, t) = r\boldsymbol{p}_0 + s\boldsymbol{p}_1 + t\boldsymbol{p}_2$便可将点$\boldsymbol{p}$表示为三角形3个顶点的加权平均值。重心坐标还有一些有趣的性质，由于本书没有涉及这些内容，因此就没有将它们一一列出。感兴趣的读者不妨对重心坐标展开进一步的研究。

C.4 平面

我们可以把一个平面视为一张无限薄、无限宽以及无限长的“三无”纸。一个平面可以用一个向量$\mathbf{n}$与平面内一点$\mathbf{p}_0$来表示。垂直于平面的向量$\mathbf{n}$（不一定具有单位长度）称为该平面的法向量（normal vector），如图C.6所示。一个平面可以将空间划分为正半空间（positive half-space）与负半空间（negative half-space）。正半空间为平面前侧的空间，所谓“平面的前侧”即为法向量指向的一侧。负半空间则为平面后侧的空间。



图C.6　通过法向量$\mathbf{n}$与平面内一点$\mathbf{p}_0$定义的平面。如果$\mathbf{p}_0$是平面内一点，当且仅当向量$\mathbf{p}-\mathbf{p}_0$正交于平面法向量，则点$\mathbf{p}$也是平面内一点

从图C.6可以看到，平面的图像是由所有满足下列平面方程（plane equation）的点$\mathbf{p}$所构成的：

$$\mathbf{n}\cdot(\mathbf{p}-\mathbf{p}_0)=0$$

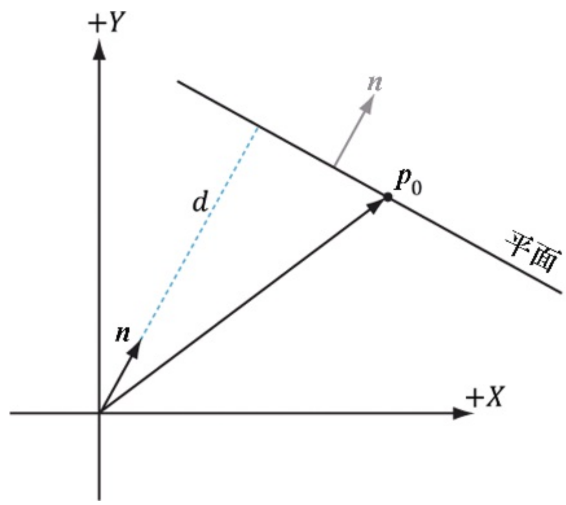
在描述一个特定平面的时候，法线$\mathbf{n}$与平面内的一已知点$\mathbf{p}_0$都是确定下来的，因此通常可以将平面方程改写为：

$$\mathbf{n}\cdot(\mathbf{p}-\mathbf{p}_0)=\mathbf{n}\cdot\mathbf{p}-\mathbf{n}\cdot\mathbf{p}_0=\mathbf{n}\cdot\mathbf{p}+d=0$$

其中$d=-\boldsymbol{n}\cdot\boldsymbol{p}_0$。如果$\boldsymbol{n}=(a,b,c)$且$\boldsymbol{p}=(x,y,z)$，那么平面方程就可以写作：

$$ax+by+cz+d=0$$

如果平面法向量$\boldsymbol{n}$为单位长度，那么$d=-\boldsymbol{n}\cdot\boldsymbol{p}_0$表示的就是从坐标系原点至此平面的最短有符号（signed，或称为“有向”）距离（见图C.7）。



图C.7　从平面到坐标系原点的最短距离

注意 Note

为了能更方便地画出示意图，我们有时绘制的是2D图像。此时，用一条直线来表示一个平面。由于一条直线可以把2D空间分为一个正半空间以及一个负半空间，所以通过一条直线以及一条与之垂直的法线就能合理地表示出2D平面。

C.4.1　DirectX数学库中平面的表示

在代码中表示平面时，存储法向量$\boldsymbol{n}$与常量d是一件很容易的事。将这两种数据看作一个4D向量是一种很不错的想法，我们可以把它们表示为$(\boldsymbol{n}, d) = (a, b, c, d)$。又因为`XMVECTOR`类型所存的正是一个浮点数值四元组，所以DirectX数学库也提供了`XMVECTOR`类型的重载用于表示平面。

C.4.2　空间点与平面的位置关系

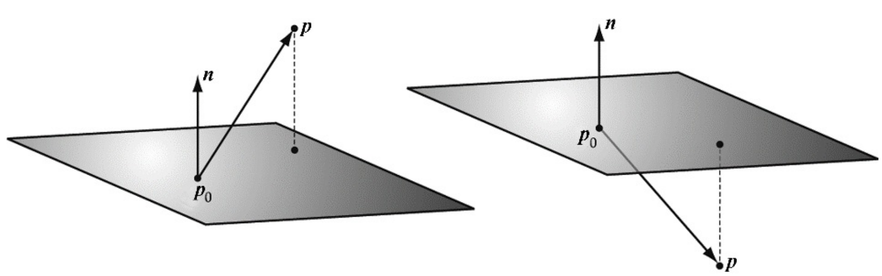
给出任意一点$\boldsymbol{p}$，从图C.6与图C.8中可以看出：

1．如果$\boldsymbol{n} \cdot (\boldsymbol{p} - \boldsymbol{p}_0) = \boldsymbol{n} \cdot \boldsymbol{p} + d > 0$，那么点$\boldsymbol{p}$位于平面的前侧空间。

2．如果$\boldsymbol{n} \cdot (\boldsymbol{p} - \boldsymbol{p}_0) = \boldsymbol{n} \cdot \boldsymbol{p} + d < 0$，那么点$\boldsymbol{p}$位于平面的后侧空间。

3．如果$\boldsymbol{n} \cdot (\boldsymbol{p} - \boldsymbol{p}_0) = \boldsymbol{n} \cdot \boldsymbol{p} + d = 0$，那么点$\boldsymbol{p}$在平面内。

这几种测试对于空间点与平面位置关系的判断是很有用处的。



图C.8　空间点与平面的位置关系

以下DirectX数学库函数用于为特定平面与点计算$\mathbf{n} \cdot \mathbf{p} + d$。

```
XMVECTOR XMPlaneDotCoord(// 返回每一个坐标分量都存有n·p+d计算结果的XMVECTOR
     XMVECTOR P, // 用平面系数(a, b, c, d)来表示的平面
     XMVECTOR V); // 用分量w = 1来表示的点

// 检测点与平面的相对位置关系
XMVECTOR p = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);

XMVECTOR v = XMVectorSet(3.0f, 5.0f, 2.0f);

float x = XMVectorGetX( XMPlaneDotCoord(p, v) );
if( x 约等于 0.0f ) // 点v在平面p内
if( x > 0 )   // 点v处于正半空间
if( x < 0 )   // 点v处于负半空间
```

注意　Note

上述代码中所说的“约等于”是由于浮点值的误差所致。

类似的函数有：

```
XMVECTOR XMPlaneDotNormal(XMVECTOR Plane, XMVECTOR Vec);
```

此函数将返回平面法向量与指定的3D向量的点积。

C.4.3　构建平面

除了直接指定平面系数$(\boldsymbol{n}, d) = (a, b, c, d)$之外，还可以用另外两种方式来计算这些系数。给出法线$\boldsymbol{n}$与平面内一已知点$\boldsymbol{p}_0$，就能解出d：

$$\boldsymbol{n} \cdot \boldsymbol{p}_0 + d = 0 \Rightarrow d = -\boldsymbol{n} \cdot \boldsymbol{p}_0$$

DirectX数学库提供的以下函数，将根据给出的平面内一点与法线来构建一个对应的平面：

```
XMVECTOR XMPlaneFromPointNormal(
     XMVECTOR Point,
     XMVECTOR Normal);
```

第二种构建平面的方式是指定目标平面内3个不共线的点。

给出平面内不共线的3点$\boldsymbol{p}_0$、$\boldsymbol{p}_1$、$\boldsymbol{p}_2$，我们就能表示此出平面内的两个向量：

$$\boldsymbol{u} = \boldsymbol{p}_1 - \boldsymbol{p}_0$$
$$\boldsymbol{v} = \boldsymbol{p}_2 - \boldsymbol{p}_0$$

据此，我们就可通过这两个平面内向量的叉积，计算出此平面的法线（回顾左手拇指法则（left hand thumb rule））：

$$\boldsymbol{n} = \boldsymbol{u} \times \boldsymbol{v}$$

接下来，我们就能计算$d = -\boldsymbol{n} \cdot \boldsymbol{p}_0$了。

相应地，DirectX数学库还提供了下列函数，用以根据用户给出的3个不共线点，计算其所在的平面：

```
XMVECTOR XMPlaneFromPoints(
     XMVECTOR Point1,
     XMVECTOR Point2,
     XMVECTOR Point3);
```

C.4.4　对平面进行规范化处理

有时候，我们可能需要对一个平面的法向量进行规范化处理。乍一想，对法向量进行规范化处理的方式似乎应当与处理其他向量的方法差不多。但是又考虑到d项的表示也依赖于法向量：$d=-\boldsymbol{n}\cdot\boldsymbol{p}_0$，因此，如果要对平面的法向量进行规范化处理，势必要重新计算d。这可以通过下述方法来实现：

$$d'=\frac{d}{\|\boldsymbol{n}\|}=-\frac{\boldsymbol{n}}{\|\boldsymbol{n}\|}\cdot\boldsymbol{p}_0$$

这样一来，我们就得到了下列对平面$(\boldsymbol{n},\ d)$的法向量进行规范化处理的公式：

$$\frac{1}{\|\boldsymbol{n}\|}(\boldsymbol{n},d)=\left(\frac{\boldsymbol{n}}{\|\boldsymbol{n}\|},\frac{d}{\|\boldsymbol{n}\|}\right)$$

另外，我们可以通过下述DirectX数学库函数来对平面的法向量进行规范化处理：

```
XMVECTOR XMPlaneNormalize(XMVECTOR P);
```

C.4.5　对平面进行变换

[Lengyel02]证明了：如果要对一个平面$(\boldsymbol{n},\ d)$进行某种变换，可以先将它当作一个4D向量，再令它乘以所需变换矩阵的逆转置矩阵即

可。注意，在变换之前，首先要保证平面的法向量必须是规范化的。可采用下列DirectX数学函数库来实现平面的变换：

```
XMVECTOR XMPlaneTransform(XMVECTOR P, XMMATRIX M);
```

示例代码：

```
XMMATRIX T(...); // 将T初始化为所需的变换矩阵
XMMATRIX invT = XMMatrixInverse(XMMatrixDeterminant(T), T);
XMMATRIX invTransposeT = XMMatrixTranspose(invT);

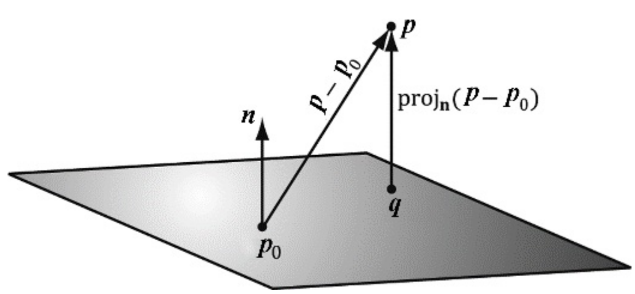
XMVECTOR p = (...); // 初始化平面
p = XMPlaneNormalize(p); // 确认法线是规范化的

XMVECTOR transformedPlane = XMPlaneTransform(p, &invTransposeT);
```

C.4.6　平面内离指定点最近的点

假设有一空间点$\boldsymbol{p}$，现欲求出平面$(\boldsymbol{n}, d)$内至点$\boldsymbol{p}$最近的一点$\boldsymbol{q}$。从图C.9可以观察到：

$$\boldsymbol{q} = \boldsymbol{p} - \mathrm{proj}_n(\boldsymbol{p} - \boldsymbol{p}_0)$$



图C.9　到点$\boldsymbol{p}$最近的平面内一点。图中的$\boldsymbol{p}_0$亦为平面内的点

假设$\|\boldsymbol{n}\| = 1$，因此有$\mathrm{proj}_n(\boldsymbol{p} - \boldsymbol{p}_0) = [(\boldsymbol{p} - \boldsymbol{p}_0) \cdot \boldsymbol{n}]\,\boldsymbol{n}$，我们就能将上

式改写为：

$$\begin{aligned} \boldsymbol{q} &= \boldsymbol{p} - [(\boldsymbol{p} - \boldsymbol{p}_0) \cdot \boldsymbol{n}]\, \boldsymbol{n} \\ &= \boldsymbol{p} - (\boldsymbol{p} \cdot \boldsymbol{n} - \boldsymbol{p}_0 \cdot \boldsymbol{n})\, \boldsymbol{n} \\ &= \boldsymbol{p} - (\boldsymbol{p} \cdot \boldsymbol{n} + d)\, \boldsymbol{n} \end{aligned}$$

C.4.7　射线与平面的相交检测

指定一条射线$\boldsymbol{p}(t) = \boldsymbol{p}_0 + t\boldsymbol{u}$与一个平面的方程$\boldsymbol{n} \cdot \boldsymbol{p} + d = 0$，我们就能知道射线与平面是否相交以及相交时的交点。为此，我们要将射线代入平面方程，并解出满足此平面方程的参数t，继而根据参数t求出两者的交点。

$\boldsymbol{n} \cdot \boldsymbol{p}(t) + d = 0$	将射线代入平面方程
$\boldsymbol{n} \cdot (\boldsymbol{p}_0 + t\boldsymbol{u}) + d = 0$	根据射线方程进行替换
$\boldsymbol{n} \cdot \boldsymbol{p}_0 + t\boldsymbol{n} \cdot \boldsymbol{u} + d = 0$	向量点积的分配律
$t\boldsymbol{n} \cdot \boldsymbol{u} = -\boldsymbol{n} \cdot \boldsymbol{p}_0 - d$	两侧同时加上$-\boldsymbol{n} \cdot \boldsymbol{p}_0 - d$
$t = \dfrac{-\boldsymbol{n} \cdot \boldsymbol{p}_0 - d}{\boldsymbol{n} \cdot \boldsymbol{u}}$	解出t

如果$\boldsymbol{n} \cdot \boldsymbol{u} = 0$，那么射线与平面平行，而方程不是无解就是有无穷多个解（如果射线在平面内，那么此方程就有无穷多个解）。如果t不

在区间 $[0, \infty)$ 内，那么射线不与平面相交，但该射线所在直线与平面相交（因为射线整体位于某半空间，背平面而行）。若t位于区间 $[0, \infty)$ 内，那么射线与平面相交，而且可以通过 t为$t_0 = \frac{-\boldsymbol{n} \cdot \boldsymbol{p}_0 - d}{\boldsymbol{n} \cdot \boldsymbol{u}}$的射线方程求出二者的交点。

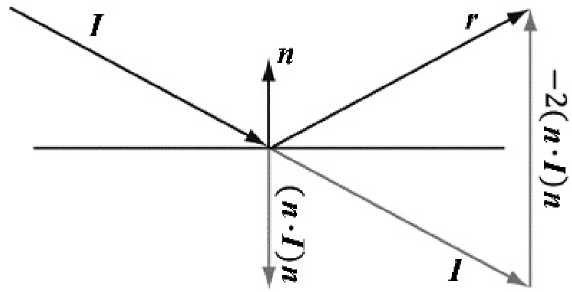
射线与平面的相交检测可以方便地改作线段与平面的相交检测。若给出定义一条线段的两个端点$\boldsymbol{p}$与$\boldsymbol{q}$，那么就可以用它们来构建射线$\boldsymbol{r}(t) = \boldsymbol{p} + t(\boldsymbol{q} - \boldsymbol{p})$。下面我们要将这条射线用于相交检测。如果$t \in [0,\ 1]$，那么线段与平面相交，否则就不相交。DirectX数学库为此而提供了以下函数：

```
XMVECTOR XMPlaneIntersectLine(
     XMVECTOR P,
     XMVECTOR LinePoint1,
     XMVECTOR LinePoint2);
```

C.4.8　反射向量

指定一个向量$\boldsymbol{I}$，欲求出它关于某平面法线$\boldsymbol{n}$的反射向量。由于向量没有位置的概念，因此只得用平面法线来参与反射向量的计算。图C.10展示了这种情况下的几何关系。此时可通过以下公式来求出反射向量：

$$\boldsymbol{r} = \boldsymbol{I} - 2(\boldsymbol{n} \cdot \boldsymbol{I})\boldsymbol{n}$$

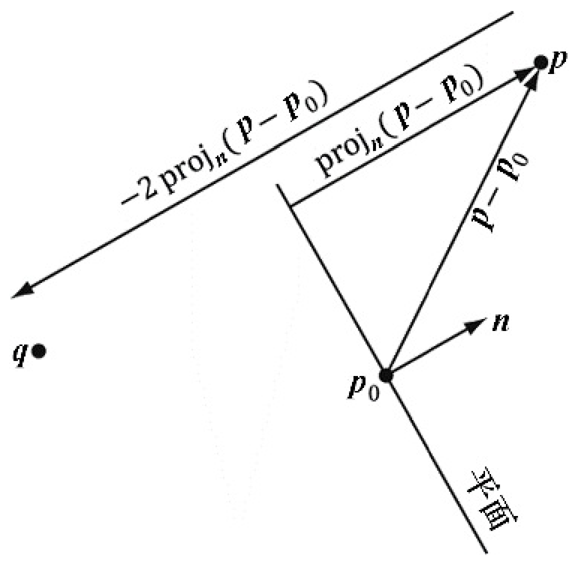


图C.10　计算反射向量的几何图示

C.4.9　反射点

由于点具有位置的概念，因此向量与点的反射是不同的。图C.11所示的反射点$\boldsymbol{q}$可以表示为：

$$\boldsymbol{q}=\boldsymbol{p}-2\mathrm{proj}_{n}(\boldsymbol{p}-\boldsymbol{p}_0)$$



图C.11　计算反射点的几何示意图

C.4.10　反射矩阵

设$(\boldsymbol{n}, d) = (n_x, n_y, n_z, d)$为一个平面的方程系数，其中$d = -\boldsymbol{n} \cdot \boldsymbol{p}_0$。接下来，利用齐次坐标，我们就能仅通过一个$4 \times 4$反射矩阵来求出关于此平面的反射点与反射向量：

$$\boldsymbol{R} = \begin{bmatrix} 1-2n_xn_x & -2n_xn_y & -2n_xn_y & 0 \\ -2n_xn_y & 1-2n_yn_y & -2n_yn_z & 0 \\ -2n_xn_z & -2n_yn_z & 1-2n_zn_z & 0 \\ -2dn_x & -2dn_y & -2dn_z & 1 \end{bmatrix}$$

利用此矩阵的前提是，假设目标平面已经过规范化处理，因此：

$$\begin{aligned} &\text{proj}_n(\boldsymbol{p}-\boldsymbol{p}_0) = -[\boldsymbol{n} \cdot (\boldsymbol{p}-\boldsymbol{p}_0)]\boldsymbol{n} \\ &= -[\boldsymbol{n} \cdot (\boldsymbol{p}-\boldsymbol{p}_0)]\boldsymbol{n} \\ &= [\boldsymbol{n} \cdot \boldsymbol{p} + d]\boldsymbol{n} \end{aligned}$$

如果令一个点与此矩阵相乘，则将得到此点的反射公式：

$$[p_x, p_y, p_z, 1] \begin{bmatrix} 1-2n_xn_x & -2n_xn_y & -2n_xn_y & 0 \\ -2n_xn_y & 1-2n_yn_y & -2n_yn_z & 0 \\ -2n_xn_z & -2n_yn_z & 1-2n_zn_z & 0 \\ -2dn_x & -2dn_y & -2dn_z & 1 \end{bmatrix}$$

$$= \begin{bmatrix} p_x - 2p_xn_xn_x & -2p_yn_xn_y & -2p_zn_xn_z & -2dn_x \\ -2p_xn_xn_y & -2p_yn_yn_y & -2p_zn_yn_z & -2dn_y \\ -2p_xn_xn_z & -2p_yn_yn_z & -2p_zn_zn_z & -2dn_z \\ & 1 & & \end{bmatrix}^{\mathrm{T}}$$

$$= \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}^{\mathrm{T}} + \begin{bmatrix} -2n_x(p_xn_x + p_yn_y + p_zn_z + d) \\ -2n_y(p_xn_x + p_yn_y + p_zn_z + d \\ -2n_z(p_xn_x + p_yn_y + p_zn_z + d \\ 0 \end{bmatrix}^{\mathrm{T}}$$

$$
= \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}^{\mathrm{T}} + \begin{bmatrix} -2n_x(\boldsymbol{n}\cdot\boldsymbol{p}+d) \\ -2n_y(\boldsymbol{n}\cdot\boldsymbol{p}+d) \\ -2n_z(\boldsymbol{n}\cdot\boldsymbol{p}+d) \\ 0 \end{bmatrix}^{\mathrm{T}}
$$

$$
= \boldsymbol{p} - 2[\boldsymbol{n}\cdot\boldsymbol{p}+d]\boldsymbol{n}
$$

$$
= \boldsymbol{p} - 2\mathrm{proj}_{n}(\boldsymbol{p}-\boldsymbol{p}_0)
$$

注意 Note

在推导的过程中，我们利用转置性质把行向量转换为列向量。这样就可以使公式的表示更为简洁，否则将得到一长串的行向量。

类似地，如果令一个向量与此矩阵相乘，则将得到此向量的反射公式：

$$
[v_x, v_y, v_z, 0]\begin{bmatrix} 1-2n_xn_x & -2n_xn_y & -2n_xn_z & 0 \\ -2n_xn_y & 1-2n_yn_y & -2n_yn_z & 0 \\ -2n_xn_z & -2n_yn_z & 1-2n_zn_z & 0 \\ -2dn_x & -2dn_y & -2dn_z & 1 \end{bmatrix} = \boldsymbol{v} - 2(\boldsymbol{n}\cdot\boldsymbol{v})\boldsymbol{n}
$$

以下DirectX数学库函数能够根据给定的平面构造出上述反射矩阵：

```
XMMATRIX XMMatrixReflect(XMVECTOR ReflectionPlane);
```

C.5 练习

1．设$\boldsymbol{p}(t)=(1,1)+t(2,1)$为一条相对于某坐标系的射线。试绘制当$t=0.0$、$0.5$、$1.0$、$2.0$以及$5.0$时该射线上的对应点。

2．设$\boldsymbol{p}_0$与$\boldsymbol{p}_1$为定义一条线段的两个端点。试证明线段方程也可以写作：当$t\in[0,\ 1]$时，$\boldsymbol{p}(t)=(1-t)\boldsymbol{p}_0+t\boldsymbol{p}_1$。

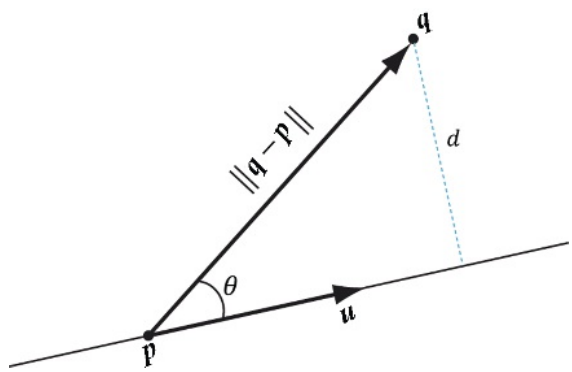
3．根据每组中的两点坐标，求取对应直线的向量直线方程。

（a）$\boldsymbol{p}_1=(2,-1)$,$\boldsymbol{p}_2=(4,1)$

（b）$\boldsymbol{p}_1=(4,-2,1)$,$\boldsymbol{p}_2=(2,3,2)$

4．设$\boldsymbol{L}(t)=\boldsymbol{p}+t\boldsymbol{u}$定义了一条3D空间中的直线。设$\boldsymbol{q}$为3D空间中的任意一点。试证明点$\boldsymbol{q}$至此直线的距离可以写作（见图C.12）：

$$d=\frac{\|(\boldsymbol{q}-\boldsymbol{p})\times\boldsymbol{u}\|}{\|\boldsymbol{u}\|}$$



图C.12　由点$\boldsymbol{q}$至目标直线的距离

5．设$\boldsymbol{L}(t)=(4,2,2)+t(1,1,1)$为一条直线。试求出下列各点至此直线的距离：

（a）$\boldsymbol{q}=(0,0,0)$

（b）$\boldsymbol{q}=(4,2,0)$

（c）$\boldsymbol{q}=(0,2,2)$

6．设$\boldsymbol{p}_0=(0,1,0)$，$\boldsymbol{p}_1=(-1,3,6)$与$\boldsymbol{p}_2=(8,5,3)$为3个点。试求出由这3点所定义的平面。

7．设$\left(\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}},-5\right)$为一个平面。定义空间点$\left(3\sqrt{3},5\sqrt{3},0\right)$、$\left(2\sqrt{3},\sqrt{3},2\sqrt{3}\right)$以及$\left(\sqrt{3},-\sqrt{3},0\right)$相对于此平面的位置关系。

8．设$\left(-\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0,\frac{5}{\sqrt{2}}\right)$为一个平面。试求出到点(0, 1, 0)最近的平面上一点。

9．设$\left(-\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0,\frac{5}{\sqrt{2}}\right)$为一个平面。试求出点(0, 1, 0)关于此平面的反射点。

10．设$\left(\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}},-5\right)$为一个平面，且$\boldsymbol{r}(t)=(-1,1,-1)+t(1,0,0)$为一条射线。试求出这条射线与上述平面的交点。接着再编写一个简短的程序，通过`XMPlaneIntersectLine`函数来验证答案的正确性。

附录D　参考资料

[Boyd10] Boyd, Chas. “DirectCompute Lecture Series 101: Introduction to DirectCompute,” 2010.

[Brennan02] Brennan, Chris. “Accurate Reflections and Refractions by Adjusting for Object Distance,” Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks. Wordware Publishing Inc., 2002.

[Burg10] Burg, John van der, “Building an Advanced Particle System.” Gamasutra, June 2000.

[Crawfis12] Crawfis, Roger. “Modern GPU Architecture.”

[De berg00] de Berg, M., M. van Kreveld, M. Overmars, and O. Schwarzkopf.

Computational Geometry: Algorithms and Applications Second Edition. Springer-Verlag Berlin Heidelberg, 2000.

[Dietrich] Dietrich, Sim. “Texture Space Bump Maps.”

[Dunlop03] Dunlop, Robert. “FPS Versus Frame Time,” 2003.

[DirectXMath] DirectXMath Online Documentation, Microsoft

Corporation.

[DXSDK] Microsoft DirectX June 2010 SDK Documentation, Microsoft Corporation.

[Eberly01] Eberly, David H., 3D Game Engine Design, Morgan Kaufmann Publishers, Inc, San Francisco CA, 2001.

[Engel02] Engel, Wolfgang (Editor), Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks, Wordware Publishing, Plano TX, 2002.

[Engel04] Engel, Wolfgang (Editor), ShaderX2: Shader Programming Tips & Tricks with DirectX 9, Wordware Publishing, Plano TX, 2004.

[Engel06] Engel, Wolfgang (Editor), ShaderX5: Shader Advanced Rendering Techniques, Charles River Media, Inc., 2006.

[Engel08] Engel, Wolfgang (Editor), ShaderX6: Shader Advanced Rendering Techniques, Charles River Media, Inc., 2008.

[Farin98] Farin, Gerald, and Dianne Hansford. The Geometry Toolbox: For Graphics and Modeling. AK Peters, Ltd., 1998.

[Fernando03] Fernando, Randima, and Mark J. Kilgard. The CG Tutorial: The Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley, 2003.

[Fraleigh95] Fraleigh, John B., and Raymond A. Beauregard. Linear

Algebra 3rd Edition. Addison-Wesley, 1995.

[Friedberg03] Friedberg, Stephen H., Arnold J. Insel, and Lawrence E. Spence. Linear Algebra Fourth Edition. Pearson Education, Inc., 2003.

[Fung10] Fung, James. “DirectCompute Lecture Series 210: GPU Optimizations and Performance,” 2010.

[Halliday01] Halliday, David, Robert Resnick, and Jearl Walker. Fundamentals of Physics: Sixth Edition. John Wiley & Sons, Inc, 2001.

[Hausner98] Hausner, Melvin. A Vector Space Approach to Geometry. Dover Publications, Inc. (www.doverpublications.com), 1998.

[Hoffmann75] Hoffmann, Banesh. About Vectors. Dover Publications, Inc., 1975.

[Isidoro06] Isidoro, John R. “Shadow Mapping: GPU-based Tips and Techniques,” Game Developers Conference, ATI slide presentation, 2006.

[Isidoro06b] Isidoro, John R. “Edge Masking and Per-Texel Depth Extent Propagation for Computation Culling During Shadow Mapping,” ShaderX 5: Advanced Rendering Techniques. Charles River Media, 2007.

[Kilgard99] Kilgard, Mark J., “Creating Reflections and Shadows Using Stencil Buffers,” Game Developers Conference, NVIDIA slide presentation, 1999.

[Kilgard01] Kilgard, Mark J. “Shadow Mapping with Today’s OpenGL Hardware,” Computer Entertainment Software Association’s CEDEC, NVIDIA presentation, 2001.

[Kryachko05] Kryachko, Yuri. “Using Vertex Texture Displacement for Realistic Water Rendering,” GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation. Addison-Wesley, 2005.

[Kuipers99] Kuipers, Jack B. Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality. Princeton University Press, 1999.

[Lengyel02] Lengyel, Eric, Mathematics for 3D Game Programming and Computer Graphics. Charles River Media, Inc., 2002.

[Möller08] Möller, Tomas, and Eric Haines. Real-Time Rendering: Third Edition. AK Peters, Ltd., 2008.

[Mortenson99] Mortenson, M.E. Mathematics for Computer Graphics Applications. Industrial Press, Inc., 1999.

[NVIDIA05] Antialiasing with Transparency, NVIDIA Corporation, 2005.

[NVIDIA08] GPU Programming Guide GeForce 8 and 9 Series, NVIDIA Corporation, 2008.

[NVIDIA09] NVIDIA's Next Generation CUDA Compute Architecture: Fermi, NVIDIA Corporation, 2009.

[NVIDIA10] DirectCompute Programming Guide, NVIDIA Corporation, 2007–2010.

[Thibieroz13] Thibieroz, Nick and Holger Gruen. "DirectX Performance Reloaded." Presentation at Game Developers Conference 2013.

[Oliveira10] Oliveira, Gustavo. "Designing Fast Cross-Platform SIMD Vector Libraries," 2010.

[Parent02] Parent, Rick. Computer Animation: Algorithms and Techniques. Morgan Kaufmann Publishers, 2002.

[Pelzer04] Pelzer, Kurt. "Rendering Countless Blades of Waving Grass," GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics. Addison-Wesley, 2004.

[Pettineo12] Pettineo, Matt. "A Closer Look at Tone Mapping," 2012.

[Petzold99] Petzold, Charles, Programming Windows, Fifth Edition, Microsoft Press, Redmond WA, 1999.

[Prosise99] Prosise, Jeff, Programming Windows with MFC, Second Edition, Microsoft Press, Redmond WA, 1999.

[Reinhard10] Reinhard, Erik, et al, High Dynamic Range Imaging,

Second Edition, Morgan Kaufmann, 2010.

[Santrock03] Santrock, John W. Psychology 7. The McGraw-Hill Companies, Inc., 2003.

[Savchenko00] Savchenko, Sergei, 3D Graphics Programming: Games and Beyond, Sams Publishing, 2000.

[Schneider03] Schneider, Philip J., and David H. Eberly. Geometric Tools for Computer Graphics. Morgan Kaufmann Publishers, 2003.

[Snook03] Snook, Greg. Real-Time 3D Terrain Engines using C++ and DirectX9. Charles River Media, Inc., 2003.

[Story10] Story, Jon, and Cem Cebenoyan, “Tessellation Performance,” Game Developers Conference, NVIDIA slide presentation, 2010.

[Sutherland74] Sutherland, I. E., and G. W. Hodgeman. Reentrant Polygon Clipping. Communications of the ACM, 17(1):32-42, 1974.

[Tuft10] Tuft, David. “Cascaded Shadow Maps,” 2010.

[Uralsky05] Uralsky, Yuri. “Efficient Soft-Edged Shadows Using Pixel Shader Branching,” GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation. Addison-Wesley, 2005.

[Verth04] Verth, James M. van, and Lars M. Bishop. Essential

Mathematics for Games & Interactive Applications: A Programmer's Guide. Morgan Kaufmann Publishers, 2004.

[Vlachos01] Vlachos, Alex, Jörg Peters, Chas Boyd, and Jason L. Mitchell, "Curved PN Triangles," ACM Symposium on Interactive 3D Graphics 2001, pp. 159-166, 2001.

[Watt92] Watt, Alan, and Mark Watt, Advanced Animation and Rendering Techniques: Theory and Practice, Addison-Wesley, 1992.

[Watt00] Watt, Alan, 3D Computer Graphics, Third Edition, Addison-Wesley, 2000.

[Watt01] Watt, Alan, and Fabio Policarpo, 3D Games: Real-time Rendering and Software Technology, Addison-Wesley, 2001.

[Weinreich98] Weinreich, Gabriel, Geometrical Vectors. The University of Chicago Press, Chicago, 1998.

[Whatley05] Whatley, David. "Toward Photorealism in Virtual Botany," GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation. Addison-Wesley, 2005.

[Wloka03] Wloka, Matthias. "Batch, Batch, Batch: What Does It Really Mean? " Presentation at Game Developers Conference 2003.